UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

SMART DATA DASHBOARDS

Alejandro Sánchez Iniesta
Tutor: Juan Antonio Óscar Barberá
Ponente: Simone Santini

Marzo de 2020

SMART DATA DASHBOARDS

AUTOR: Alejandro Sánchez Iniesta
TUTOR: Juan Antonio Óscar Barberá

Dpto. Ingeniería informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Marzo de 2020

Resumen (castellano)

El trabajo de fin de grado centra su atención en el desarrollo de una aplicación Java para la obtención y simulación, explotación y representación de datos, tanto de forma gráfica en su representación como a nivel de informes, teniendo como objetivo principal permitir visualizar al usuario en una sola pantalla y a tiempo real, la abstracción de una cantidad voluminosa de datos de una manera visual y representativa.

La aplicación admite múltiples orígenes de datos con diferente formato, almacenados en BD relacionales o multidimensionales, o recogidos de otros orígenes (documentos de texto, hojas de cálculo, archivos CSV, XML, etcétera).

Una vez definidos y organizados estas fuentes de datos, la aplicación procede a su normalización para su posterior explotación, jugando para ello con “ontologías” de forma que, independientemente de su origen, podamos invocar a todas las entidades que puedan intervenir en estas representaciones gráficas, de forma persistente.

De esta forma, para construir estos tableros (dashboards), tendremos a nuestra disposición una serie de entidades (gadgets de representación de las ontologías) que podremos arrastrar y soltar a nuestros tableros (a modo de kpis) y poder cruzar y relacionar dichas entidades y la representación gráfica de dichas relaciones. Dichos tableros estarán alimentados por una serie de señales, que podrán suministrar datos de manera analógica (con cierta granularidad), digital (sin granularidad determinada). Ambos tipos de señales podrán combinarse en un mismo dashboard.

La potencia de dicha aplicación radica en la capacidad de esta para establecer diferentes capas de abstracción en relación con diferentes parámetros, tales como el origen de los datos, su naturaleza, su trazabilidad y linaje, su historificación y su formato, atacando una serie de procesos como la ingesta de forma masiva de cualquier tipo de información almacenada, su normalización para su posterior tratamiento y su conversión a ontologías para su explotación, tanto a nivel gráfico como textual.

Abstract (English)

This Bachelor Thesis focuses on the development of a Java application that obtains, simulates, exploits and represents data, both graphically in its representation and at the level of reports, with the main goal of allowing the user to visualize on a single screen and in real time, the abstraction of a big quantity of data, in a visual and representative way.

The application supports multiple data sources with different formats, stored in relational or multidimensional databases, or collected from other sources (text documents, spreadsheets, CSV files, XML, etc.).

Once these data sources are defined and organized, the application proceeds to its normalization for later exploitation, playing with “ontologies” so that, regardless of their origin, we can invoke all the entities that can intervene in these graphic representations, persistently.

In this way, to build these dashboards, we will have at our disposal a series of entities (gadgets representing the ontologies) that we can drag and drop to our boards (as kpis) and be able to cross and relate these entities and the graphic representation of these relationships. These boards will be filled by a series of signals, which can supply data analogically (with a certain granularity) or digitally (without determined granularity). Both types of signals can be combined on the same dashboard.

The power of this application lies in its ability to establish different layers of abstraction in relation to different parameters, such as the origin of the data, its nature, its traceability and lineage, its historification and its format, attacking a series of processes such as the massive intake of any type of stored information, its normalization for its subsequent treatment and its conversion to ontologies for its exploitation, both graphically and textually.

Palabras clave (castellano)

Dashboard, gestión de datos, ingesta de datos, Java, ontología, gadget, timeserie, diseño, desarrollo.

Keywords (inglés)

Dashboard, data management, data intake, Java, ontology, gadget, timeserie, design, development.

Agradecimientos

A mis compañeros de la universidad por hacer de la carrera algo ameno y estar dispuestos a ayudar cuando era necesario.

A mis compañeros del trabajo por darme esta oportunidad y su disposición a ayudarme en todo lo que he necesitado.

A mi familia por darme apoyo en todo momento.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE CÓDIGOS

1 Introducción

En este apartado se va a realizar una introducción que nos permita conocer el motivo por el que este proyecto está teniendo lugar, además de resumir los principales objetivos que pretende abarcar.

1.1 Motivación

Esta memoria de trabajo de fin de grado tiene como objetivo explicar los diferentes pasos del proceso del desarrollo de la aplicación Java de "Smart Data Dashboards", una aplicación que permite crear tableros dinámicos para la visualización de datos de manera rápida y representativa.

En la actualidad, la importancia de los datos se ha incrementado hasta el punto de ser uno de los principales temas a tratar por la mayoría de las empresas, desde gestión de bases de datos hasta el famoso Big Data. Esta importancia reside principalmente en el incremento del volumen de los datos disponibles hoy en día, además de las nuevas tecnologías orientadas a reunir, almacenar y analizar los mismos. Se calcula que en los últimos dos años se han generado más datos que en todo el resto de la historia de la humanidad, y que en la actualidad cada ser humano produce una huella digital de 1,7MB por segundo [1]. Con tal volumen de información, la creación de herramientas que analicen, aglutinen o visualicen dichos datos, cobran cada vez más importancia.

La posesión de datos permite a las empresas tomar decisiones acerca de sus productos de manera objetiva y, por tanto, ser creados acorde a un nicho de mercado ideal, o mejorarlos gradualmente para que se adapten cada vez mejor al mundo real. Estos datos son suministrados por diferentes fuentes (a veces comprados), como por ejemplo dispositivos móviles, historiales de búsqueda de un navegador, datos sectoriales recopilados por empresas especializadas... pudiendo ser tipos de datos estructurados, no estructurados, o definiciones intermedias. [2]

Todos estos datos anteriormente mencionados, no servirían de gran cosa si no pueden ser interpretados. En la mayoría de los casos, cuando los datos son utilizados por una aplicación, como por ejemplo un motor de búsqueda, será un algoritmo el que se encargue de realizar una interpretación fiel a la realidad y sacar las conclusiones oportunas y necesarias para el correcto funcionamiento de esta. En otros casos, no es necesaria la interpretación para esta obtención de soluciones, sino que es necesaria una representación sencilla a modo de gráfica, mapa... para dar una visión general del aspecto de los datos a un público en concreto. Esta rápida visualización puede ser utilizada con fines expositivos, para realizar una supervisión a tiempo real de una determinada señal, revisar rápidamente la tendencia de los datos… y demás ocasiones en las que visualizar datos directamente desde una BD o un documento podría resultar una tarea más complicada y tediosa, o requerir cierto procesamiento.

La visualización de los datos permite a los usuarios captar conceptos complejos que no se verían de otra manera, identificar patrones temporales,... Estas interpretaciones son muy

importantes a la hora de comunicar conclusiones dentro de un equipo de trabajo, donde una herramienta como la que se va a desarrollar permite generar informes visuales que respalden una nueva idea ante la futura decisión del equipo.

Un ejemplo perfecto de aglutinamiento de una cantidad de datos muy voluminosa resumida en una sola imagen es la figura 1-1, donde se representa la rutina diaria de los estadounidenses divida en las diferentes actividades que desempeñan a lo largo del día. [3]

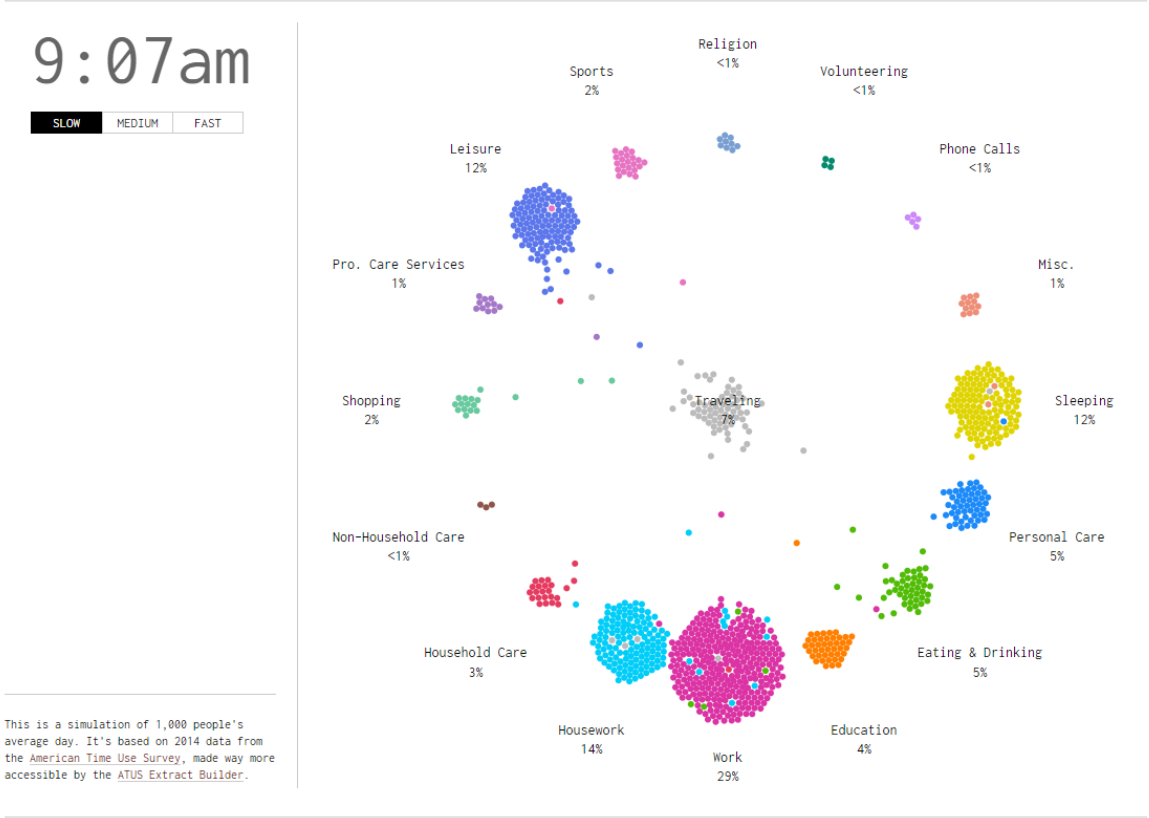

Figura 1-1: A day in the live of Americans

Cada punto representa a una persona, que se mueve dinámicamente haciendo la simulación de un día completo. En conjunto, tenemos una representación de la rutina de 1000 personas en una sola pantalla.

En Minsait se ha surgido la necesidad de tener a su disposición una herramienta que disponga una visualización de datos en forma de tableros que pueda ser utilizada por diferentes productos, y que, además, haga uso de las estructuras de datos y ontologías de la Onesait Platform (OP) [4], una plataforma de funcionalidades que es transversal a la mayoría de los productos de la empresa. Al tener un diseño orientado a ser usado por productos que utilicen la OP, y a su vez, estar utilizando estas herramientas para su propio desarrollo, resultará muy sencillo integrar esta aplicación en un proyecto en marcha. Esto dota a la aplicación de cierto acoplamiento a la gama de productos de Minsait, pero dado que los potenciales usuarios de la aplicación son equipos de trabajo de dichos productos, y que la línea de pensamiento de la empresa pretende centralizar la mayor parte de la implementación de un producto dentro de la OP, es un factor que resulta de poca importancia.

1.2 Objetivos

A continuación, se resumirán los principales objetivos que la aplicación deberá de cumplir al ser finalizada:

Obj-1. La aplicación debe ser capaz de crear, visualizar, editar, importar, exportar y eliminar los dashboards alojados en el sistema.

Obj-2. Los dashboards serán dotados de un nombre, un propietario, y una serie de gadgets que se almacenan en su interior, así como otros campos de descripción, estado y configuración.

Obj-3. La aplicación utilizará un módulo de inicio de sesión que permita a un determinado usuario visualizar una serie de dashboards específicos a los que tiene acceso, quedando ocultos para aquellos que no tengan permisos para los mismos.

Obj-3.1. El usuario tendrá a su disposición un apartado de perfil de usuario donde puede visualizar su nombre de usuario, su rol dentro de la instancia del grupo de usuarios en el que se encuentra, entre otros datos.

Obj-4. La aplicación debe poder ser integrable en cualquier proyecto de Minsait que haga uso de Onesait Platform en cuestión de un tiempo razonable, tras realizar las configuraciones pertinentes en las variables de entorno que apunten a la máquina deseada.

Obj-5. La aplicación pondrá a disposición del usuario entre 5 y 10 gadgets diferentes que permitan tener diversas opciones a la hora de representar un dato. De esta manera se podrá elegir el gadget que mejor represente las cualidades deseadas.

Obj-5.1 Los gadgets generados tendrán un diseño simple e intuitivo, que permita al usuario realizar un breve análisis a simple vista y sacar unas conclusiones rápidas.

Obj-6. Los gadgets deben ser capaces de representar dos tipos de señales: las señales analógicas, que serán las que obtienen un valor cada cierto intervalo de tiempo de manera regular (normalmente guardadas en timeseries), y las señales digitales, que son datos que pueden no estar relacionados temporalmente.

Obj-7. El proyecto utilizará el sistema de Continuous Integration/Continuous Delivery (CI/CD) utilizado en la empresa, que permite disponibilizar las versiones de la aplicación automáticamente tras la subida a GIT.

Obj-8. La interfaz gráfica de la aplicación (Frontend) debe tener el mismo “flavour” y estructuración que el resto de los módulos y herramientas del entorno de Minsait.

1.3 Organización de la memoria

A continuación, se detallará la estructura que sigue la memoria a lo largo de sus apartados. Se ha decidido hacer una estructuración acorde con el proceso de desarrollo software incremental, coincidiendo cada capítulo con un paso del proceso.

La memoria consta de los siguientes capítulos:

- **Capítulo 2: Estado del arte.** Capítulo en el que se detalla cuáles son las opciones en el momento actual si se quisiese hacer uso de una aplicación u otro sistema similar que pueda ser una opción frente a esta aplicación.
- **Capítulo 3: Diseño.** Capítulo en el que se especificarán los detalles de diseño tomados para la aplicación, ya sea de su funcionalidad, interfaz, o detalles estructurales.
- **Capítulo 4. Desarrollo.** Capítulo en el que se detallarán los aspectos relacionados con la implementación los sistemas de la aplicación.
- **Capítulo 5. Integración, pruebas y resultados.** Capítulo en el que se explicará el flujo CI/CD seguido y los tests que la aplicación ha tenido que superar, entre otras pruebas.
- **Capítulo 6. Conclusiones y trabajo futuro.** Capítulo en el que se cerrará la memoria, con unas conclusiones basadas en la satisfacción de los objetivos propuestos al inicio del proceso software, y los próximos pasos que se podrían dar en el desarrollo de la aplicación.

2 Estado del arte

En este apartado se procede a realizar un análisis comparativo entre las diferentes opciones que existen actualmente en el mercado que ofrecen una funcionalidad similar a la que se ha propuesto para los Smart Dashboards. Se analizarán las ventajas e inconvenientes que caracterizan a cada una de las opciones, y como podrían resolver la necesidad que existe en la empresa para la visualización de datos. Se estudiarán factores como la facilidad de utilización, la estética, la obtención de fuentes de datos o la capacidad de personalización.

Analizando las opciones más populares, vemos que las herramientas más utilizadas son aplicaciones web, normalmente con una versión gratuita o de prueba, y una versión de pago con funcionalidad adicional. Además de estas, existen otras herramientas que no están orientadas en exclusiva a la creación de dashboards, pero podrían ser una opción válida debido a su potencia, como es el caso de Excel. La principal desventaja que se puede encontrar en todas ellas puede ser la dependencia de un tercero en términos de disponibilidad, y la escasa o nula integración en un proyecto. Otro inconveniente muy común es la realización de un paso intermedio para poner a disposición de estas aplicaciones un datasource que, inicialmente, se encontraba dentro de la Onesait Platform. Esto es, realizar una exportación de los datasources a una BD local, para después ser importados en la aplicación en cuestión.

A continuación, se exponen las mejores opciones encontradas. [5]

2.1 Chart.io.

Se trata de una aplicación web que permite la creación de dashboards, no solo orientados a un entorno de trabajo, sino a un nivel más coloquial y al alcance de cualquiera. Sus principios son la flexibilidad a la hora de obtener las fuentes de datos, la mejora de la colaboración y el trabajo en equipo mediante a una herramienta en común y la rapidez en la creación de dashboards.

2.1.1 Ventajas

- Facilidad a la hora de generar los gadgets.
- Diseño sencillo y limpio de cada uno de los tipos de gráficas.
- Versión completa gratuita durante 14 días.

2.1.2 Inconvenientes

- Requiere un registro previo.
- Versión de pago a partir del periodo de prueba.
- Poca variedad en los gráficos a elegir.

2.2 Cyfe.

Cyfe es una aplicación web que permite monitorizar datos de una forma dinámica y profesional. Da a elegir diferentes dashboards orientados al sector al cual te dediques, lo que permite una mayor adaptabilidad de los gadgets a las necesidades de cada usuario. Destaca entre la competencia por tener funcionalidades exclusivas como la adaptación de dashboards al tamaño de un monitor, reportes en tiempo real o servicio de alertas vía email.

2.2.1 Ventajas

- Versión con funcionalidad reducida gratuita.
- División de tipos de dashboards por sector o mercado.
- Posibilidad de representar datos en tiempo real.

2.2.2 Inconvenientes

- Poca flexibilidad dentro de la personalización de cada gadget.
- Limitaciones de la versión gratuita.

2.3 KlipFolio

Klipfolio es una aplicación web que se dedica a la creación de dashboards para la visualización de datos a nivel de usuario. Permite realizar una monitorización de cuentas, sobre todo de redes sociales, clientes de correo y otras herramientas, para ver a simple vista el estado de tus redes. A diferencia del resto, está orientada a llevar una organización interna de la información de un usuario en vez de centrarse en la puesta en común de datos en un entorno de trabajo.

2.3.1 Ventajas

- Permite el registro con cuenta de Google rápidamente.
- Gran variedad de gadgets predefinidos.
- Datasources procedentes de redes sociales fáciles de configurar.

2.3.2 Inconvenientes

- Datasources procedentes de bases de datos requieren configuración adicional.
- No posee una versión de prueba gratuita.
- Poca polivalencia si trabajamos en sectores ajenos a redes sociales.

2.4 Microsoft Office Excel

Aunque no es la primera opción que se nos puede venir a la cabeza, la increíble potencia de Excel nos permite crear algo muy parecido a lo que consideramos dashboards en la aplicación a desarrollar. Pone a disposición del usuario todas las utilidades del ámbito de las matemáticas, estadística y representación de datos a los que estamos acostumbrados, además de diversas extensiones y plugins disponibles, con el inconveniente del tiempo que se tendrá que emplear para conseguir un resultado

2.4.1 Ventajas

- Pone a disposición del usuario todo tipo de personalización de gadgets.
- Facilidad de obtención de datos (.csv, bases de datos SQL).

2.4.2 Inconvenientes

- Para lograr un buen resultado la complejidad puede ser alta.
- Requiere el uso de Windows como sistema operativo.
- Dificultad para visualizar datos a tiempo real.

3 Diseño

3.1 Estructura del proyecto

En este apartado dentro del bloque de diseño, se van a detallar los aspectos relativos a la estructura general del proyecto, los diferentes componentes que formarán la estructura, y la comunicación que existirá entre ellos.

Teniendo en cuenta la necesidad de lanzar una aplicación lo más reutilizable posible, es conveniente dividir la funcionalidad en módulos, que puedan modificarse por separado e incluso, poder ser integrados sin el resto de los módulos en un proyecto que no tenga la necesidad de utilizar todos ellos. Por esta razón se va a hacer uso de microservicios, es decir, dividir la aplicación completa en varias partes que se comportan como servicios independientes, pueden ser actualizados o desplegados por separado, y su funcionamiento no depende de los demás. [6]

Ligado a esto último, se pretende utilizar el patrón MVC (Modelo-Vista-Controlador) para permitir la implementación independiente de cada módulo, garantizando así su actualización y mantenimiento de forma segura. [7].

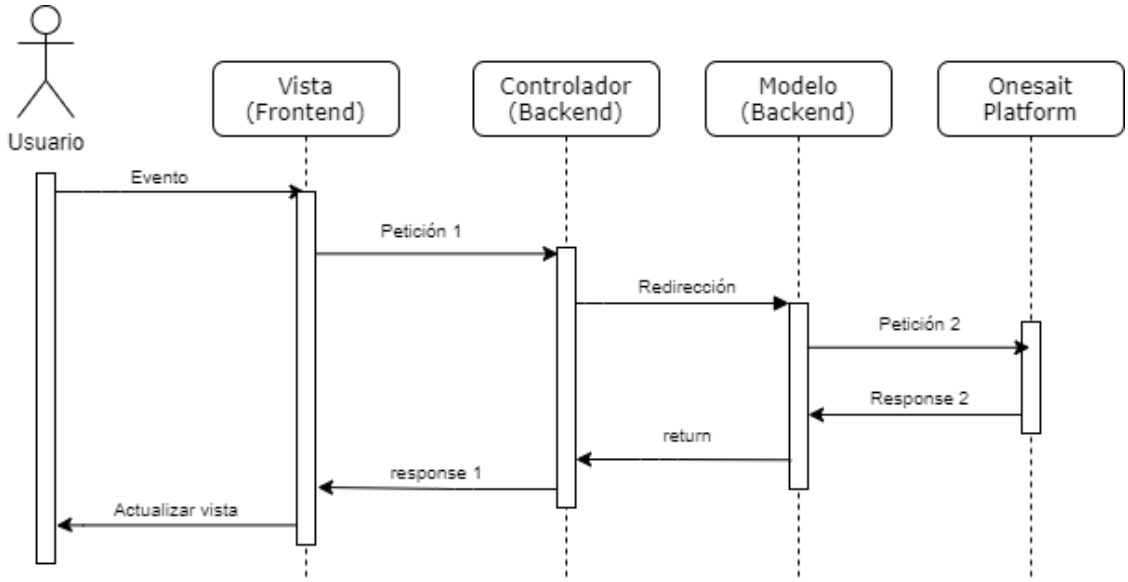

Figura 3-1: Diagrama de secuencia de una petición a la aplicación

Como podemos observar en el esquema de la figura 3-1, el modelo está implementado en el Backend, ya que es el que contiene las estructuras de datos y los servicios que se encargan de las transformaciones dentro de los mismos. La vista sería llevada a cabo por el módulo de Frontend, que recoge los datos del modelo (que a su vez fueron obtenidos de la OP mediante peticiones REST), y rellena una vista con los mismos. La funcionalidad de controlador también la encontramos en una parte del Backend, donde varias clases hacen la función de controladores para recibir las diferentes peticiones y comunicar los datos recibidos a uno de los servicios.

La aplicación en sí está compuesta por los módulos de Backend y Frontend. El módulo de Backend se encarga de realizar toda la gestión de estructuras de datos, especificación de modelos, configuración de despliegue y de securización, etc. El módulo de Frontend es el encargado de generar una vista para parte de la funcionalidad que el Backend ofrece, centrándose en acciones como el inicio de sesión, la visualización del perfil de usuario, o la gestión y visualización de los propios dashboards. Parte de la funcionalidad del Backend, como la creación de nuevas plantillas de gadgets o gestión de cuentas de usuarios, se gestionará directamente desde una API que realizará las peticiones oportunas a la Onesait Platform. El esquema general que se deduce del funcionamiento de la aplicación es el siguiente:

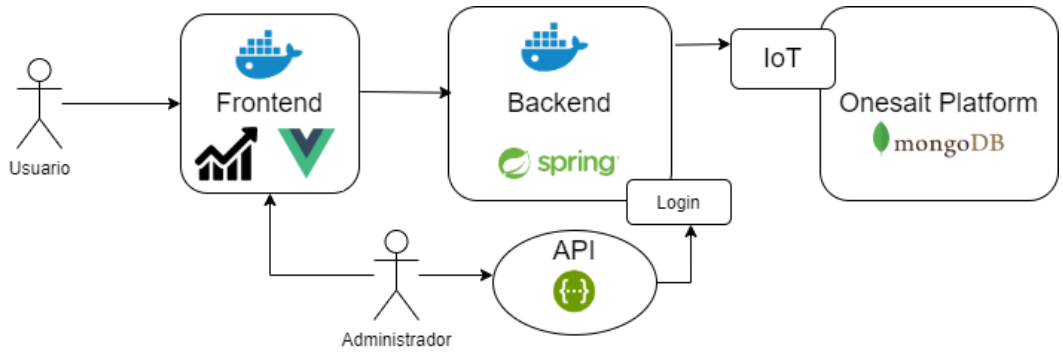

Figura 3-2: Estructura simplificada de los módulos de la aplicación

Como se puede apreciar en el esquema de la figura 3-2, el Backend y el Frontend se comunican a través de peticiones que realiza el Frontend que demanda información, y, en Backend a su vez, realiza peticiones a la OP para la actualización de los datos. La OP se utiliza en la mayoría de las ocasiones como sistema de almacenamiento para todas las estructuras de datos necesarias, aunque también es la que se encarga de almacenar usuarios en el realm y dividir a estos según su rol. La conexión con la OP se consigue gracias a la configuración de un cliente IoT alojado en la aplicación, que permitirá el acceso a las ontologías necesarias. La API permite a los administradores acceder a funcionalidad limitada haciendo peticiones directas al Backend.

3.2 Funcionalidad de la aplicación

En este apartado se definirá toda la funcionalidad que la aplicación debe abarcar en su versión final. Para simplificar la implementación se divide la funcionalidad en sistemas, agrupando en ellos aquellas utilidades que estén relacionadas. Los sistemas están compuestos por subsistemas, que agruparán en módulos aún más específicos aquellas funciones que afectan, por ejemplo, a un modelo de datos determinado. En el siguiente esquema, los sistemas están coloreados de amarillo, mientras que los subsistemas se colorean en azul.

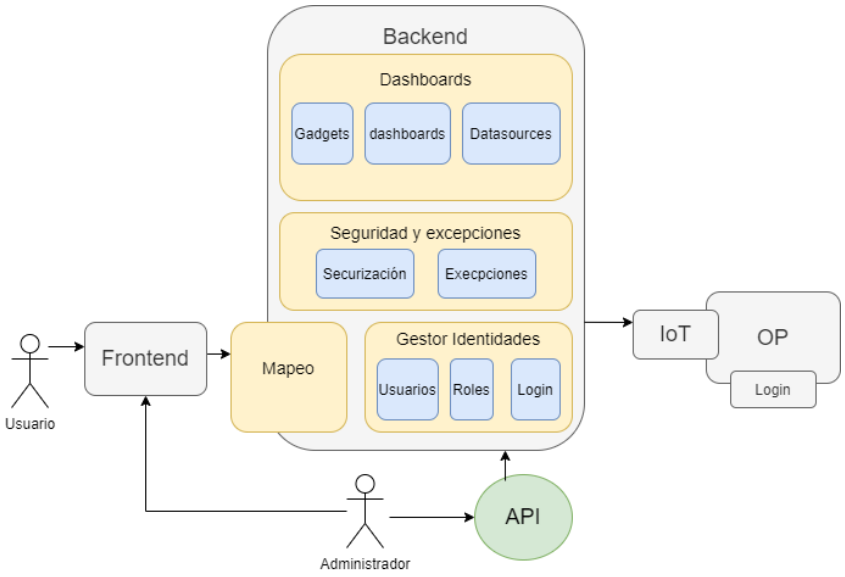

Figura 3-3: Estructura de la aplicación dividida en sistemas

Sistema de Dashboards: El sistema de dashboards engloba toda la funcionalidad necesaria para llegar a representar un dashboard por pantalla. Esto es, aquellas acciones relacionadas con la creación, edición y eliminación de los tableros, la disposición y la búsqueda de los tableros en el menú principal, o la adición de gadgets a los mismos.

- **Subsistema de Gadgets:** En este subsistema encontramos todas las utilidades necesarias para la creación de gadgets, selección de tipos, modificación del código, redimensionamiento, desplazamiento o edición del estilo de estos.

- **Subsistema de Datasources:** Este subsistema contiene la funcionalidad relacionada con la obtención de datos de las fuentes externas a la aplicación, ingesta de dichas fuentes, e introducción de los datos en las diferentes señales.

- **Subsistema de Dashboards:** Este subsistema tiene incluida la funcionalidad relacionada con la creación de dashboards gracias a todos los elementos descritos anteriormente.

Sistema de Gestión de Identidades: Este sistema engloba toda la funcionalidad que tenga que ver con las formas de acceso a la aplicación, la creación, modificación de características y eliminación de los usuarios que pueden acceder al sistema, así como los diferentes permisos que estos tienen, y permiten realizar o no determinadas acciones.

- **Subsistema de Inicio de sesión:** Este subsistema se encarga de las acciones necesarias para tener un sistema de “login” para proteger la aplicación de posibles accesos indeseados, y las medidas de seguridad que este componente utilice.

- **Subsistema de Gestión de usuarios:** Este subsistema contiene toda aquella funcionalidad relacionada con la creación, obtención, edición, o eliminación de los usuarios de la aplicación.

- **Subsistema de Roles:** Este subsistema contiene toda aquella funcionalidad relacionada con la adjudicación de los diferentes roles a los usuarios, la limitación

de acciones que estos conllevan para los usuarios, así como la creación de nuevos roles si fuesen necesarios.

Sistema de Mapeo de modelos: Este sistema contiene la funcionalidad relacionada con la obtención de estructura de datos de la OP, y mapeo, edición y eliminación de atributos para la creación de instancias de clases (DTO's) en nuestra aplicación, que se ajusten a las necesidades de estas.

Sistema de seguridad y excepciones: Este sistema engloba toda la funcionalidad relacionada con la securización de la aplicación y sus endpoints y datos, así como la gestión que se realiza de los errores que puedan darse en la ejecución de esta y el tratamiento a través de excepciones.

- **Subsistema de securización:** Este subsistema se encarga de realizar el bloqueo y las redirecciones pertinentes entre endpoints en la aplicación, con la finalidad de limitar el acceso a aquellos usuarios que no estén dados de alta en el sistema.

- **Subsistema de Excepciones:** Este subsistema es el encargado de gestionar cualquier flujo incorrecto de operaciones que acabe en error, mediante a la generación y el uso de excepciones propias que definan de una manera clara las causas por las que ciertas acciones no se han podido llevar a cabo.

3.2.1 Requisitos funcionales

En esta sección se van a detallar los requisitos funcionales que la aplicación debe cumplir al final del proceso software, es decir, aquellas funciones que la aplicación o sus módulos deben desempeñar para cumplir con los objetivos planteados. Para una mejor ordenación de los requisitos, se van a ordenar según al sistema al que pertenezcan indicado en el punto 3.2.

3.2.1.1 Sistema de dashboards

R.F-1 Creación de dashboards: La aplicación deberá de dar la opción al usuario de crear dashboards desde el menú principal (si posee permisos para ello), introduciendo un nombre para el mismo desde una pantalla emergente.

R.F-2 Edición de dashboards: La aplicación deberá de dar la opción al usuario de editar los dashboards desde la vista del dashboard o desde el menú principal (si posee permisos para ello), permitiendo introducir, editar o eliminar sus gadgets.

R.F-3 Eliminación de dashboards: La aplicación deberá de dar la opción al usuario de eliminar dashboards desde el menú principal (si posee permisos para ello), pasando por una ventana emergente de confirmación.

R.F-4 Búsqueda y ordenado de dashboards: La aplicación deberá de dar la opción al usuario de realizar una búsqueda por nombre u ordenar los mismos por orden alfabético o fecha de creación, todo ello desde una barra de búsqueda y un desplegable en el menú principal.

R.F-5 Importación y exportación de dashboards: La aplicación deberá de dar la opción al usuario de exportar sus dashboards en formato JSON dashboards desde el menú principal (si posee permisos para ello), para posteriormente ser exportados en

otra instancia de la aplicación desde el menú principal y poder ser visualizados por otros usuarios.

3.2.1.1.1 Subsistema de gadgets

R.F-6 Creación de gadgets: En la vista de un dashboard, el usuario con permisos de edición debe ser capaz de crear gadgets mediante a el arrastre de una plantilla desde un menú desplegable con todas las plantillas disponibles.

R.F-7 Eliminación de gadgets: En la vista de un dashboard, el usuario con permisos de edición debe ser capaz de eliminar gadgets mediante a una opción en el menú interior de cada gadget.

R.F-8 Edición de gadgets: En la vista de un dashboard, el usuario con permisos de edición debe ser capaz de editar gadgets mediante a una opción en el menú interior de cada gadget, que permitirá cambiar el estilo de este.

R.F-9 Redimensionamiento de gadgets: En la vista de un dashboard, cualquier usuario debe ser capaz de redimensionar un gadget para verlo en pantalla completa, y si tiene los permisos necesarios, aumentar su tamaño en relación con los demás dentro del dashboard.

3.2.1.1.2 Subsistema de Datasources

R.F-10 Generación de datos: La aplicación debe tener algún método de simulación de datos que inserte en las ontologías de la OP los datos necesarios para generar señales que puedan visualizarse mediante gadgets. Este método debe dar a elegir al usuario la ventana de tiempo que ocupa la señal, así como su granularidad.

R.F-11 Obtención de datos: El sistema debe ser capaz de asimilar todas aquellas señales guardadas en la ontología de la OP, sin importar la fuente de donde procede, siempre y cuando mantengan el formato exigido por la OP.

3.2.1.2 Sistema de gestión de identidades

3.2.1.2.1 Subsistema de inicio de sesión

R.F-12 Inicio de sesión: La aplicación debe poseer la funcionalidad necesaria para iniciar sesión con un usuario y una contraseña creados previamente por un administrador dentro del realm al cual la aplicación apunte.

R.F-13 Datos únicos: La aplicación debe tener en cuenta que algunos campos privados del usuario, como su nombre o su correo electrónico deben ser únicos dentro del realm, al intentar crear un usuario con el mismo nombre que uno existente.

R.F-14 Mantener sesión: El sistema dará una opción desde el menú de inicio de sesión para mantener la sesión iniciada mediante a la persistencia de un token, que durará hasta que el usuario cierre sesión o el token caduque.

R.F-15 Soporte cambio contraseñas: El sistema dará una opción desde el menú de inicio de sesión para obtener los datos necesarios del servicio de soporte de la máquina, para poder contactar con un administrador que autorice el cambio de contraseñas.

3.2.1.2.2 Subsistema de gestión de usuarios

R.F-16 Perfil de usuario: El usuario será capaz de entrar a una pantalla de perfil de usuario desde el menú principal, la cual le permitirá visualizar datos como su usuario, email, nombre completo o rol dentro de la aplicación.

R.F-17 Creación de usuarios: Un administrador debe ser capaz de crear usuarios mediante el uso de la API (vía Swagger).

R.F-18 Actualización de usuarios: Un administrador debe ser capaz de modificar los atributos de los usuarios mediante el uso de la API (vía Swagger), entre los que se incluyen su contraseña o su rol en la aplicación.

R.F-19 Eliminación de usuarios: Un administrador debe ser capaz de eliminar usuarios mediante el uso de la API (vía Swagger).

3.2.1.2.3 Subsistema de roles

R.F-20 Gestión de permisos: El sistema debe ser capaz de, dados unos roles determinados, prohibir o permitir el acceso a determinadas funcionalidades dependiendo de la jerarquía de estos.

R.F-21 Adjudicación de roles: El administrador deberá adjudicar un rol determinado a un usuario en el momento de su creación.

R.F-22 Gestión de roles: El administrador podrá modificar el rol de un usuario mediante la API para adjudicarle nuevas funcionalidades o quitárselas.

3.2.1.3 Sistema de mapeo de modelos

R.F-23 Mapeo de modelo-DTO: La aplicación debe implementar la funcionalidad necesaria para, una vez obtenidos los modelos de datos suministrados por la OP, realizar el mapeo de atributos necesarios para crear estructuras propias con la estructura deseada (DTO´s).

R.F-24 Mapeo de DTO-modelo: La aplicación debe implementar la funcionalidad necesaria para convertir los DTO´s utilizados por la aplicación de vuelta en modelos de datos soportados por la OP.

3.2.1.4 Sistema de seguridad y excepciones

3.2.1.4.1 Subsistema de securización

R.F-25 Redirección a Login: El sistema será capaz de detectar si se está intentando acceder a una URL prohibida para usuarios no identificados, y redirigirá esta petición a la pantalla de inicio de sesión para que el usuario se identifique.

R.F-26 Ocultamiento de endpoints: El sistema debe ser capaz de responder a ciertas peticiones con un 403 cuando un usuario sin los permisos necesarios intente acceder a un determinado endpoint que de acceso a una funcionalidad limitada.

3.2.1.4.2 Subsistema de excepciones

R.F-27 Gestión de códigos de estado: La aplicación dispondrá de una serie de clases para lanzar una excepción personalizada cuando una de las peticiones devuelva uno de los códigos de estado de error más frecuentes (400, 401, 500…).

R.F-28 Gestión de errores internos: La aplicación dispondrá de una serie de clases para lanzar una excepción personalizada cuando se obtenga un error al crear, modificar o eliminar alguno de los componentes más utilizados (dashboard, señales, roles…).

3.2.2 Requisitos no funcionales

A continuación, se van a especificar los requisitos no funcionales que la aplicación debe tener en cuenta durante su desarrollo.

R.NF-1: Transición entre vistas: el tiempo de carga máximo que puede haber entre una pantalla y otra no deberá exceder los 3 segundos. En el caso de la carga de la vista del dashboard, todos los gadgets deben ser cargados en un tiempo no superior a 6 segundos.

R.NF-2: Adaptabilidad de pantallas: La interfaz debe ajustarse correctamente a la resolución de cualquier monitor. La aplicación debe ajustar su interfaz a la pantalla de una Tablet cuando se esté accediendo a través de uno de estos dispositivos.

R.NF-3: Limitación de memoria RAM: Una vez desplegada en una máquina la aplicación no deberá de exceder los 600Mb de RAM (incluyendo Frontend y Backend) cuando esta se encuentre a pleno funcionamiento.

R.NF-4: Multilenguaje: La aplicación poseerá la opción de cambiar el lenguaje entre español e inglés desde el menú principal.

R.NF-5: "Flavour" de la capa Frontend: La aplicación debe usar una interfaz acorde con la gama de productos para los que pretende ser útil. Esto es, debe utilizar una serie de componentes, colores, fuentes y tamaños de letra, iconos… predeterminados por la gama de productos Onesait.

R.NF-6: Privacidad de usuarios: El administrador o administradores son los únicos usuarios que pueden tener acceso al perfil de los usuarios de la aplicación. Los usuarios sin este rol no podrán tener acceso a información personal de otros usuarios.

R.NF-7: Integrabilidad: La aplicación debe diseñarse para poder ser integrable en el tiempo de una jornada laboral, al menos en el 50% de los productos. Para todos aquellos productos que sigan con la normativa de estructuración y despliegue de la empresa, la aplicación debe ser integrable en el 90% de los casos en una jornada laboral.

R.NF-8: Calidad de código: La aplicación debe superar los estándares de calidad de la empresa de manera satisfactoria según el análisis desarrollado mediante la herramienta SonarQube.

R.NF-9: Accesibilidad de la API: La aplicación debe ser accesible mediante a una API con un diseño y una documentación generados por las herramientas proporcionadas por Swagger y Swagger.ui. Esta puede ser pública pero solo podrá ser utilizada por los administradores gracias a la securización mediante tokens de seguridad.

R.NF-10: Tests de la aplicación: El sistema de dashboard debe de poseer cobertura de Tests proporcionada por la herramienta Mockito.

3.3 Modelo de clases

A continuación, se da una representación del diagrama de clases que pretende servir como muestra para el desarrollo de la aplicación.

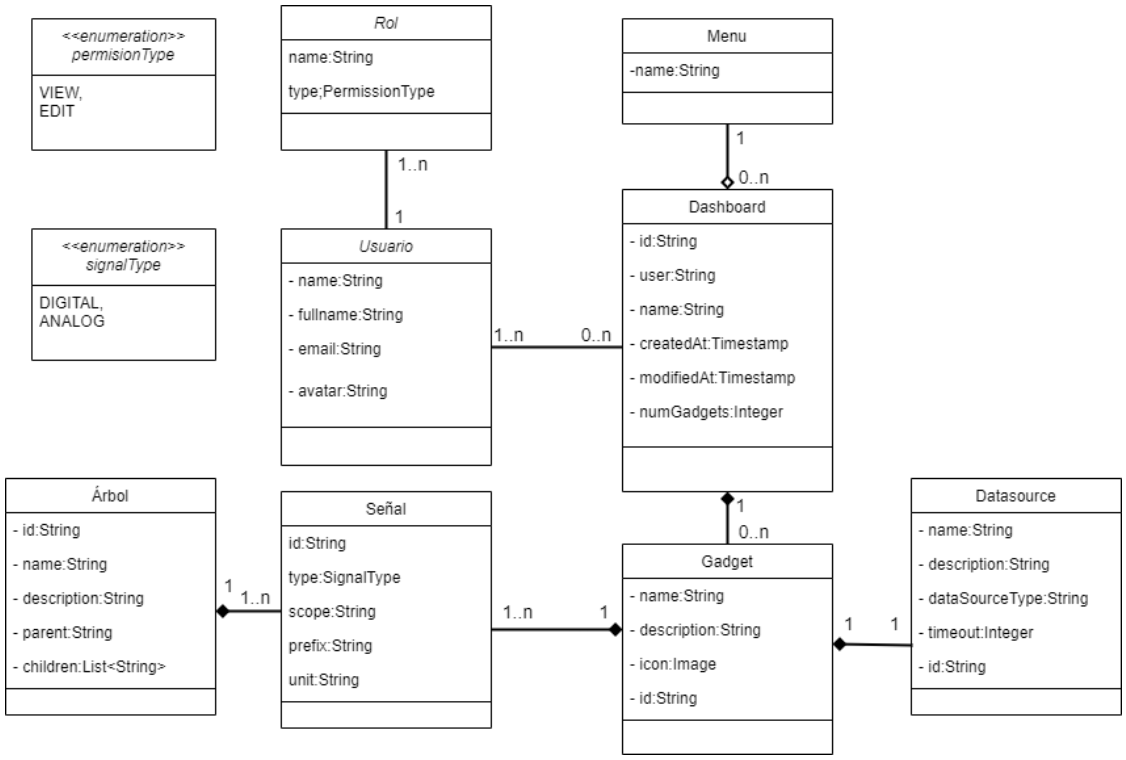

Figura 3-4: Diagrama de clases de la aplicación

Tomando como referencia el diagrama de la figura 3-4, se detallan las siguientes clases:

- **Dashboard:** contiene todos los datos y metainformación necesaria de los tableros. Vemos que se compone de gadgets, y pueden ser vistos por una serie de usuarios.
- **Usuario:** contiene los datos del usuario que se mostrarán en su perfil, y puede tener acceso a varios dashboards, y debe tener al menos un rol asignado.
- **Rol:** establece una serie de roles mediante un enumerado (permissionType), que marcará que permisos tiene el rol en cuestión.
- **Menú:** compuesto de ninguno, uno, o varios dashboards.
- **Gadget:** clase que engloba todos los tipos de gadgets que se pueden generar, todos con los mismos atributos de cara a la aplicación. Están alimentados por un datasource, y al menos por una señal.
- **Datasource:** clase que describe el origen de datos de un gadget. Normalmente se referirá a una ontología de la OP en su atributo "dataSourceType".
- **Señal:** Estructura de datos que contiene una serie de información, ordenada según su atributo "type", que se determina mediante una enumeración.
- **Árbol:** clase que ordena las señales en forma de árbol. En su creación, las señales determinan cuál es su padre, y estas se guardan jerarquizadas en la OP, por lo que la implementación de esta clase en la aplicación no es necesaria.

4 Desarrollo

Este capítulo tratará con todo lo relacionado con el proceso de desarrollo que se ha llevado a cabo, empezando por describir las herramientas y tecnologías utilizadas, siguiendo con la estructuración de los módulos y el desarrollo de los sistemas, y, por último, la implementación de las diferentes conexiones y relaciones que permiten el flujo de información entre módulos.

4.1 Herramientas y tecnologías de desarrollo

En esta sección se va a dar explicación de todas las tecnologías, lenguajes de programación y herramientas que se han utilizado a lo largo del desarrollo de la aplicación, y los motivos por los cuales han sido elegidas.

Como lenguaje de programación, he escogido Java 8 debido a mi experiencia personal con el lenguaje, además de por su versatilidad, la orientación a objetos y la cantidad de librerías disponibles. Java sigue siendo el lenguaje de programación más usado para el desarrollo de aplicaciones web [8], por lo que la cantidad de fuentes de consulta es muy grande. Como Framework de Java, se utiliza Spring Framework [9] por su soporte a nivel de aplicación y la configuración que ofrece a la hora de estructurar y desplegar el proyecto Java, además de las facilidades que ofrece a la hora de programar, aportando un nivel superior del que obtenemos con Java. Dadas todas estas facilidades, el uso de Spring se ha incrementado en los últimos años hasta ser el Framework de Java, llegando incluso a competir con frameworks como Django (Python). En la figura 4-1 podemos ver una comparativa del uso de dichos frameworks a lo largo de los últimos años. [10]

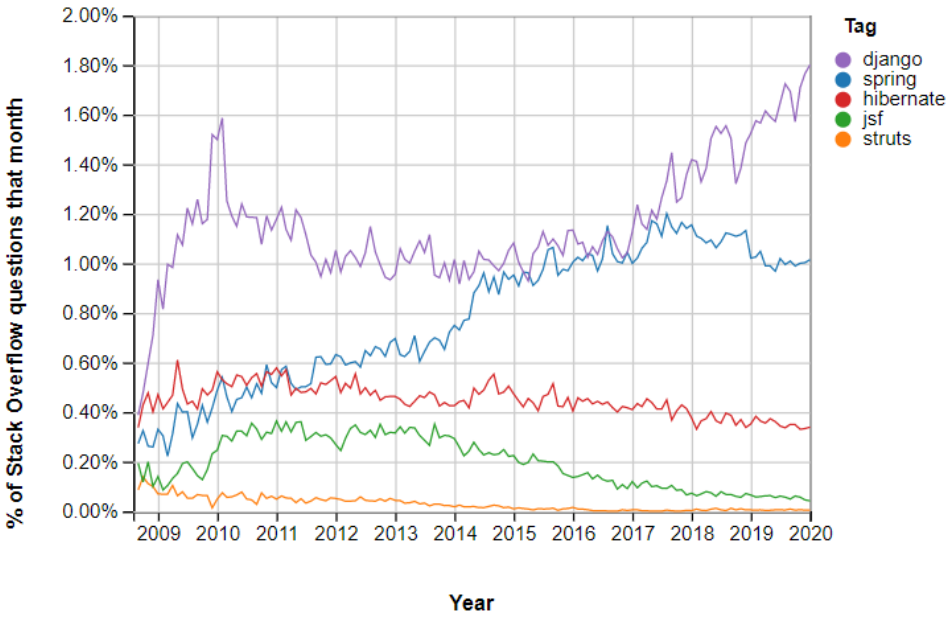

Figura 4-1: Frecuencia en % de búsquedas de etiquetas de Stack Overflow de los principales frameworks de Java

Ligado a este último, se utiliza STS (Spring Tool Suite) como entorno de programación, que proporciona herramientas de ejecución de tests, debug o búsqueda y formateo de texto, entre otras opciones.

Pasando al desarrollo de la API REST, se ha utilizado Swagger y Swagger.ui, para proporcionar a los usuarios administradores todas las facilidades posibles en términos de documentación y visualización de las peticiones, para que realice las llamadas a la aplicación de la manera más sencilla posible.

En la parte de Frontend, se ha utilizado JavaScript como lenguaje de programación, y Vue.js como framework, para reducir la complejidad del código y obtener una capa de abstracción por encima de lo que JavaScript propone. Estos últimos han sido impuestos por la política de la empresa, pero esto no quita que la combinación mencionada sea de las más utilizadas en el desarrollo de frontales web [10]. La siguiente comparación muestra el crecimiento del uso de Vue.js frente a otros frameworks como React, más conocidos en el sector tecnológico.

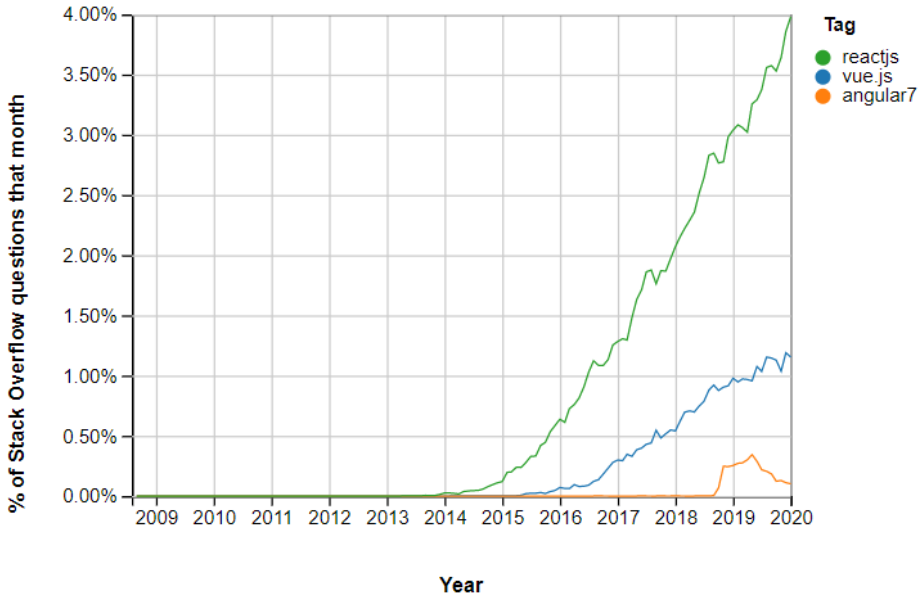

Figura 4-2: Frecuencia en % de búsquedas de etiquetas de Stack Overflow de los principales frameworks de JavaScript

Adicionalmente, se ha utilizado ECharts, una librería de visualización de gráficas muy potente, que nos permitirá generar las plantillas de los gadgets a partir de una estructura JSON bastante simple.

Por último, y respecto al almacenamiento de datos, se utilizará la Onesait Platform como almacén de las estructuras generadas en la aplicación. Estas, (una vez almacenadas, llamadas ontologías) se guardan en bases de datos de MongoDB que podrán ser consultadas desde la API o desde una herramienta en la plataforma en caso de tener dicha necesidad.

4.2 Estructura del proyecto Backend

En esta sección, se pretende dar una visualización de como se ha organizado la estructura del proyecto a nivel de paquetes, diferenciando estos entre la funcionalidad que cada uno lleva a cabo. Esta estructuración viene dada por las directrices que la empresa exige a la hora de estructurar un proyecto, para, una vez pasado el proceso de desarrollo, los pasos de despliegue e integración sean sencillos dada su compatibilidad con otros módulos con la misma forma.

A continuación, se da un esquema que retrata el árbol de paquetes llevado a cabo en la aplicación. Todos los paquetes cuelgan de la ruta ***com.minsait.onesait.architecture.smartdashbord*** por convención de la empresa, y después se ha seguido una estructuración acorde con los subsistemas y lo que es habitual en una aplicación web. En la figura 4-3 podemos apreciar en detalle la estructuración del proyecto.

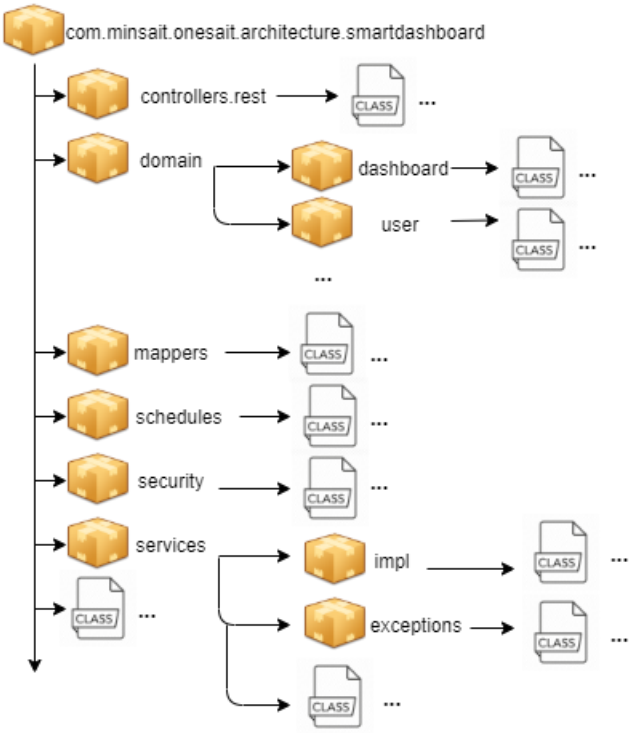

Figura 4-3: Estructura de paquetes de la aplicación

En el directorio principal encontramos la clase principal de la aplicación, la configuración de Swagger.ui, la configuración del protocolo SSL, y algunas utilidades comunes a todos los paquetes subyacentes.

Bajo el paquete **controllers.rest** se encuentran todas las clases que realizan la función de controlador, e implementar aquellos endpoints que se van a disponibilizar a través de la API, o bien son llamados directamente desde la parte Frontend del proyecto. Estos endpoints reciben como parámetros identificadores o modelos de datos que serán redireccionados a los servicios para realizar las funciones de creación, obtención y modificación de las bases de datos.

En el paquete **domain** tenemos agrupados todos aquellos directorios que contienen las clases de las distintas estructuras de datos, tanto del sistema de dashboards (dashboards, gadgets, Datasources…) como el sistema de gestión de identidades (usuarios y roles). Cada uno de los directorios posee diferentes clases y extensiones de estas, que son versiones diferentes

del objeto que servirán para mostrar una vista simplificada de uno de los ellos, o bien todas sus características.

En el paquete **mappers** están todas aquellas clases que se utilizan para convertir las estructuras de datos de la aplicación en estructuras interpretables por la OP, y viceversa.

En el directorio **schedules** se encuentra una clase (ScheduledTasks.java) que se encarga de ejecutar las tareas programadas periódicamente por la aplicación. Un ejemplo de estas tareas es la actualización de la previsualización de los dashboards desde el menú principal cada cierto tiempo.

En el paquete **security** se encuentra la clase que configura Spring Security.

En el paquete **services** podemos encontrar las interfaces de todos los servicios que se van a utilizar, y en dos subdirectorios diferentes, las implementaciones de dichos servicios en la carpeta **impl**, y todas las excepciones que se lanzan desde la aplicación en la carpeta **exceptions**.

4.3 Desarrollo de los sistemas

En esta sección se va a dar una descripción de los detalles más representativos del desarrollo de la aplicación, dividida en sistemas. Para empezar, se describirán algunos detalles que son transversales a todos los sistemas y son de suma importancia.

Se puede empezar comentando el punto de entrada a la aplicación, es decir, los controladores. Para simplificar el código de estos, se ha implementado un controlador abstracto (AbstractRestController.java) que proporciona a todos los demás la gestión de excepciones diferencia los distintos códigos de error, del cliente (4xx), del servidor (5xx) u otros. En el código 4-1 podemos ver la función que se encarga de dicha utilidad.

```
protected ResponseEntity<Object> defaultException(Exception e, String message) {
	log.debug(message, e);
	if (e instanceof HttpClientErrorException) {
		HttpClientErrorException eHttp = (HttpClientErrorException) e;
		return
ResponseEntity.status(eHttp.getStatusCode()).body(eHttp.getResponseBodyAsString());
	} else if (e instanceof HttpServerErrorException) {
		HttpServerErrorException eHttp = (HttpServerErrorException) e;
		return
ResponseEntity.status(eHttp.getStatusCode()).body(eHttp.getResponseBodyAsString());
	} else {
		return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(e.getMessage());  }
}
```

Código 4-1: Función para la gestión de excepciones

Pasando a las extensiones del controlador abstracto, se puede observar que todos siguen una estructura muy similar. En primer lugar, poseen la etiqueta @RestController de Spring que marca la clase como controlador. Después, etiquetas como @Api o @ApiResponses que darán información en la interfaz gráfica de Swagger, y dividen todos los endpoints en diferentes secciones para ser más accesibles. Ya en el interior de la clase se marca con la etiqueta @Autowired una instancia del servicio al cual se va a atacar desde el controlador

en cuestión, y acto seguido se dan las diferentes funciones que realizan las llamadas al servicio. Normalmente, estas funciones crean, eliminan, actualizan, u obtienen las instancias del objeto al cual el controlador está dedicado (ya sean dashboards, gadgets, usuarios…). Para una mejor visualización de dichas funciones se va a mostrar un ejemplo sencillo de estas en la figura 4-2.

```
@ApiOperation(value = "Returns all the Dashboards")
@ApiResponse(code = 200,
             message = "All dashboards are returned",
             response = DashboardDTO[].class)
@GetMapping
public ResponseEntity<Object> getAllDashboards(
       @RequestHeader String authorization,
       @RequestParam(required = false) DashboardOrder order,
       @RequestParam(required = false) boolean includeImages) {

       order = order == null ? DashboardOrder.IDENTIFICATION_ASC : order;
       List<DashboardDTO> allDashboards =
dashboardService.getAllDashboards(authorization, order);
       if (includeImages) {
              allDashboards = dashboardService.getDashboardImages(allDashboards);
       }
       return ResponseEntity.ok(allDashboards);
}
```

Código 4-2: Función de obtención de dashboards

Las etiquetas @ApiOperation y @ApiResponses dan información de la propia función para el Swagger. La etiqueta @GetMapping indica el método HTTP con el que se realiza la petición, en este caso GET (en otros casos también incluiría la extensión del endpoint al que se quiere acceder, si este fuese distinto a la ruta base). La función recibe un token de autorización por cabeceras (obtenido después de un inicio de sesión), el orden en el que se quieren recibir los dashboards y un booleano que marca si se quieren recibir o no sus representaciones en miniatura. Posteriormente, se realiza una llamada al servicio donde se incluyen todos los dashboards en una lista, y esta lista es devuelta dentro de una ResponseEntity de tipo OK (200) como respuesta de la petición.

4.3.1 Sistema de dashboards

A continuación, se van a describir algunos detalles de la implementación del sistema de dashboards. Este sistema es bastante amplio, puesto que contempla muchas estructuras necesarias para la gestión de los dashboards, por ejemplo, los gadgets, Datasources, señales... para todas estas clases se sigue el mismo procedimiento que se explicará a continuación.

Como se ha comentado en el apartado 4.3, las peticiones llegan a la aplicación por medio de una serie de controladores. Estos controladores hacen llamadas a métodos de una instancia del servicio correspondiente. Los servicios son interfaces que listan las funciones que deben ser desarrolladas en una clase que las implemente. En el directorio “Impl” se encuentran las implementaciones de dichos servicios. Estas implementaciones son marcadas con la etiqueta @Service de Spring que indica que la clase es un bean de la capa de negocio. Posteriormente, se marcan con @Override todas las funciones a implementar, que obtienen del controlador la información necesaria para realizar las operaciones, y hacen una petición a una URL de la OP donde se guardará la información oportuna. A continuación, en el código 4-3 vemos un ejemplo de una función de un servicio, en el que se recibe información sobre un dashboard para hacer una petición HTTP y crear el mismo en la OP.

```
@Override
public DashboardDTO createDashboard(String authorization,
                                    DashboardInformationDTO informationDTO) {
        HttpHeaders httpHeaders =
        ServiceUtils.createDefaultHttpAuthHeaders(authorization);

        Map<String, Object> mvm = new LinkedHashMap<>();
        mvm.put(COMMAND, "newDashboard");
        informationDTO.setDashboardType(DashboardType.DASHBOARD);
        mvm.put("information", informationDTO);

        final HttpEntity<Map<String, Object>> httpEntity = new HttpEntity<>(mvm,
                                                                httpHeaders);
        return template.postForEntity(managementURL,
                                      httpEntity,
                                      DashboardDTO.class).getBody();

        }
```

Código 4-3: Función de servicio de creación de dashboards

Se puede observar cómo se crea la petición mediante a la introducción de una cabecera de autorización, la creación de un mapa con los datos del dashboard para insertarlo en el cuerpo de la petición, y la especificación de la URL a la cual queremos realizar la petición. La respuesta que devolvemos al controlador será el cuerpo de la respuesta obtenida, que contiene los datos del dashboard creado. El controlador se encarga de devolver un código 200 en este caso al Frontend (o a la API) si la operación ha sido exitosa. En las operaciones en las que el Frontend solicita información, por ejemplo la función que devuelve todos los dashboards existentes, además del código de estado se devuelve un objeto JSON, o una lista de objetos JSON que este utilizará para rellenar sus componentes, y mostrar de esta manera, una vista de las estructuras solicitadas.

4.3.2 Sistema de gestión de identidades

En este apartado se van a mostrar algunos detalles técnicos sobre el desarrollo del sistema de gestión de usuarios y roles. La aplicación requiere de ciertas medidas de seguridad, puesto que al estar desplegada en lo que podría ser una IP pública, podríamos recibir peticiones de usuarios indeseados. Por esta razón se hace uso de un módulo de login que proporciona el acceso a usuarios que estén registrados previamente en la OP. Además, dentro de la OP puede haber usuarios involucrados en proyectos diferentes, totalmente independientes entre sí, por lo que también hay que controlar que usuarios tienen acceso a una serie de dashboards determinados. Por último, dentro de un mismo proyecto, es posible que no todos los usuarios puedan modificar un dashboard, sino que solamente puedan visualizarlo, por lo que también hay que controlar los permisos dentro de un mismo dashboard.

El primer nivel de seguridad (acceso únicamente para usuarios de la OP) se controla mediante un login, que pide unas credenciales y son redireccionados a la OP para comprobar si estos son válidos, por lo que nuestra aplicación se desentiende de este nivel, ya que es la OP la que mantiene la lista de usuarios que tienen acceso al sistema.

El segundo nivel de seguridad (acceso según pertenencia a un proyecto) se consigue mediante realms. Un realm es un grupo de usuarios que poseen acceso a unos recursos determinados en común. Dentro de la OP puede haber diferentes realms, lo cual es útil a la hora de dar acceso a recursos de una manera limitada, en vez de a todos los usuarios de manera global. La aplicación posee unas variables de configuración que permiten atacar a

un realm específico, lo que facilita que un usuario vea solo los dashboards que le interesan en el menú principal. En el código 4-4 podemos ver la configuración del fichero application.yml para atacar a un realm determinado.

```
openplatform:
    baseUrl: https://XX.XX.XX.XX # IP donde la OP esté desplegada
    security:
      tokenVerification: true
      login:
        url: ${dashboard.baseURL}/oauth-server/oauth/token
        grant_type: password
        clientId: realmDashboard # nombre del realm
        scope: openid
        user: administrator # administrador de la OP
        password: *********
```

Código 4-4: Configuración de la aplicación contra un realm

Debido a esta distinción entre realms, un usuario no puede acceder a dos dashboards creados en diferentes realms en el mismo menú principal. La solución sería tener dos instancias de la aplicación, una para cada realm al que se quiere acceder, e introducir al usuario en ambos realms. El usuario podrá acceder a ambas instancias desplegadas contra la misma IP con endpoints diferentes.

El tercer nivel de seguridad es el que se refiere a los roles que se dan dentro de un mismo realm. Es posible que no se quiera dar permisos de edición a todos los usuarios, y que este privilegio esté limitado a algunos administradores. Dentro de un realm existe el rol de "user", con permisos de visualización, y el rol "admin", con permisos de edición, y creación de dashboards. Además de estos tipos de usuarios, existe un administrador de plataforma que es el encargado de crear los realms, aunque esta operación se realiza fuera de la aplicación. A continuación, se da un esquema del funcionamiento del sistema de realms y roles en la figura 4-4.

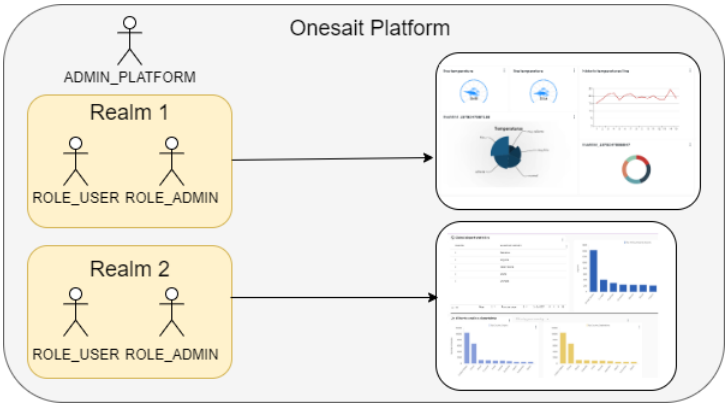

Figura 4-4: Gestión de permisos en la OP

Una vez la configuración contra un realm es realizada correctamente, que realizar la securización de determinados endpoints por medio de etiquetas. Con la etiqueta @PreAuthorize se puede especificar el rol que se debe tener dentro del realm para hacer una petición concreta. Dentro de la etiqueta especificamos parámetros como @RolSecurity.isAdmin, que limita el acceso a todo aquel que no sea administrador, o @Rolsecurity.isSameUser, que permite el acceso si las operaciones a realizar involucran al usuario activo en ese momento. Otra etiqueta utilizada es @RolSecurity.isAuthenticated,

que permitirá el acceso a cualquier usuario que haya iniciado sesión. En el código 4-5 vemos un ejemplo de estas etiquetas para la función de creación de usuarios, operación llevada a cabo por administradores.

```
@PreAuthorize("@rolSecurity.isAdmin()")
public ResponseEntity<Object> createUser(@RequestHeader String authorization,
                @Valid @RequestBody UserSimplified user) {
        try {
                service.createNewUser(authorization, user);
        } catch (DashboardUserManagementException e) {
                return ResponseEntity.status(e.getStatus()).body(e.getMessage());
        }
        return ResponseEntity.status(HttpStatus.CREATED).body(null);
}
```

Código 4-5: Función de creación de usuarios

4.3.3 Sistema de mapeo de modelos

A continuación, se dará la explicación de como se ha desarrollado el sistema que se encarga de obtener estructuras de datos de la OP, y convertirlas en clases que se adecúen al funcionamiento de nuestra aplicación, y viceversa.

La OP es una herramienta muy potente, que ofrece una cantidad de información muy grande acerca de las estructuras de datos que se están manejando. El problema viene cuando para un uso más simple como el que se da en esta aplicación, cierta información es innecesaria y debe ser ignorada. El primer paso es desarrollar una clase con la estructura de la ontología del objeto en cuestión. Esta posee un id interno y metainformación, como puede ser la fecha de la inserción o la última vez que ese objeto fue consultado. De la ontología se obtiene lo que se llama el objeto DTO, aquella clase que sirve para agrupar y almacenar los atributos, y transportarlos de una parte a otra de la aplicación de manera eficiente. En la mayoría de los casos, para realizar una operación en concreto, (por ejemplo, enviar datos de un Dashboard al Frontend) la mayoría de los atributos del DTO no se utilizan, por lo que se crean múltiples modelos de datos para cada una de las utilidades. Además, es muy posible que los nombres de los atributos cambien, o el tipo de los mismos requiera una modificación. De todas estás modificaciones se encargarán las clases alojadas en el paquete “mappers”. En la figura X.X se da una visión general de las transformaciones explicadas anteriormente.

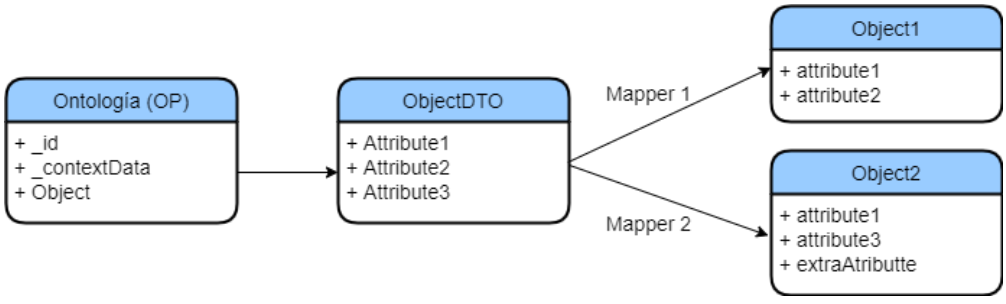

Figura 4-5: Flujo de conversiones de objetos en la aplicación

A continuación, vemos un ejemplo de una clase que realiza el mapeo de un Dashboard. Esta debe ser marcada con la etiqueta @Mapper, y cada una de las operaciones de mapeo con la etiqueta @Mapping. Como se aprecia en el código 4-6, vamos a pasar de unos datos alojados en la clase DashboardDto.java, a rellenar una nueva clase llamada DashboardResult.java. Se realiza el renombramiento de algunos atributos, se da valor a otros constantes, y se especifica el formato de fecha que usará.

```
@Mapper
public abstract class DashboardMapper {

	protected static final DateTimeFormatter FORMATTER =
		DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss.S");

	@Mapping(target = "grid", constant = "FREE-GRID")
	@Mapping(target = "numGadgets", source = "ngadgets")
	@Mapping(target = "createdAt", expression =
		"java( java.time.LocalDateTime.parse(t.getCreatedAt(),FORMATTER) )")
	@Mapping(target = "id", source = "identification")
	public abstract DashboardResult dashboardDtoToResult(DashboardDTO t);

}
```

Código 4-6: Clase para mapear objetos de DashboardDTO a DashboardResult

El proceso de pasar de una de las clases modelo de la aplicación a una instancia de la ontología es mucho más sencillo. Simplemente, se rellenan los atributos del objeto DTO con los valores del modelo, y este se envía en una petición a la OP, ya que este ya puede ser interpretado. La misma OP se encarga de establecer un id único a la nueva instancia de la ontología, al igual que generar la metainformación guardada en "contextData".

4.3.4 Sistema de seguridad y excepciones

4.3.4.1 Desarrollo del subsistema de seguridad

El framework de Spring proporciona una de sus herramientas llamada Spring Security, que proporciona diversos servicios para la securización de aplicaciones Java y acceso con autorización. Permite al usuario la configuración de dicha seguridad mediante una clase de configuración, y abstrae a este de codificaciones más complejas.

Para configurar Spring Security creamos una clase "SpringSecurityConfig.java" marcada con la etiqueta @Configuration, y que extiende de la clase de configuración "WebSecurityConfigurerAdapter.java". Se deben extender varias funciones para configurar cada una de las utilidades que se desean. Por ejemplo, para la configuración del filtro CORS, se crea un bean con los métodos y cabeceras permitidos. Para la configuración de la seguridad HTTP, se extiende una función "configure" a la que indicamos la protección a nivel de recursos. Extendiendo la función "configure" a nivel de seguridad web, protege el acceso a ciertos endpoints que son privados, y no se desea que un usuario acceda a estos desde el exterior. A continuación, en el código 4-7, se detallan dichas extensiones de la función "configure".

```
@Override
protected void configure(HttpSecurity http) throws Exception {
	http.headers().frameOptions().disable();

	http.csrf().ignoringAntMatchers("/actuator/**").and().authorizeRequests()
		.requestMatchers(EndpointRequest.to("status","info")).permitAll();

	http.csrf().disable().authorizeRequests().antMatchers("/","/home",
							"/favicon.ico").permitAll()
			.antMatchers("/api/*").permitAll();
}
@Override
public void configure(WebSecurity web) throws Exception {
web.ignoring().antMatchers("/resources/**", "/static/**", "/css/**",
				"/js/**","/images/**", "/webjars/**");
}
```

Código 4-7: Configuración de Spring Security

En la primera función se están ignorando las peticiones que contengan "/actuator/**", y se deshabilita la protección CSRF para algunos endpoints. En la segunda función se está prohibiendo el acceso a todos los recursos bajo los endpoints indicados como argumentos.

4.3.4.2 Desarrollo del subsistema de excepciones

A continuación, se explicará cómo se han desarrollado las excepciones personalizadas para manejar los posibles errores internos o peticiones erróneas que se puedan dar en la aplicación. Las excepciones se dividen en dos tipos, las que saltan cuando se da un código de estado inesperado para el usuario (400, 401, 403, 404…), que son devueltas causadas por errores al hacer la petición por parte del usuario, y aquellas que saltan cuando una operación contra la OP no ha salido como se esperaba (normalmente se devuelve un 500 ya que se considera un error interno de la aplicación, ya que es esta la que hace la petición a la OP). En el código 4-8 vemos un ejemplo de excepción del primer tipo.

```
@ResponseStatus(value = HttpStatus.BAD_REQUEST)
public class BadRequestException extends RuntimeException {

	private static final long serialVersionUID = 1L;

	public BadRequestException(String message) {
		super(message);
	}
}
```

Código 4-8: Excepción de BadRequest

Como podemos observar, este tipo de excepción, en este caso la que salta al recibir un BadRequest, extiende del tipo RuntimeException. En la etiqueta @ResponseStatus se especifica el valor del código de estado a devolver. A continuación, en el código 4-9 apreciamos una excepción del segundo tipo.

```
public class DashboardRolException extends Exception {

	private static final long serialVersionUID = 1L;
	@Getter
	private final HttpStatus status;


	public DashboardRolException(HttpStatus status, String message) {
		super(message);
		this.status = status;
	}
}
```

Código 4-9: Excepción de DashboardRolException

Podemos observar que las excepciones de este tipo extienden directamente de la clase "exception", y en este caso que se capturan los errores relacionados con la gestión de roles, se suministra un código de estado HTTP, y un mensaje que aportará una descripción más detallada del error, que se suministrará en el lugar desde donde la excepción haya sido lanzada.

4.4 Módulos auxiliares de la aplicación

En esta sección se describen dos proyectos auxiliares que han sido de utilidad para la integración y el uso de la aplicación. En un entorno de producción, estos módulos no tendrían por qué ser utilizados, por lo tanto, se han dejado fuera de la estructuración del proyecto. Estos módulos son el Config-init, que sirve para crear las estructuras de datos que serán usados por la aplicación, y DataSimulator, un simulador de datos que hace inserciones en la estructura de Timeseries con intención de realizar pruebas previas al entorno de desarrollo.

4.4.1 Módulo de Config-init

Como ya se ha adelantado, la principal funcionalidad de este módulo es acelerar la creación de estructuras de datos en la OP en el caso de no existir. Otra de sus funcionalidades es la creación de un cliente con el mismo nombre que se especifique en el application.yml de la parte Backend. En el código 4-10 se aprecian los pasos que la aplicación sigue para la creación de las ontologías.

```
@PostConstruct
public void execute() throws URISyntaxException, IOException {

	Login loginPlatform = loginPlatform(user, password);
	checkUsers.check(loginPlatform);
	loginPlatform = loginPlatform(checkUser, checkPassword);

	if (checkRealms)
		checkRealm.check(loginPlatform);

	List<String> ontologies = checkOntologies.check(loginPlatform);
	checkIndex.check(loginPlatform);
	clientPlatform.check(loginPlatform, ontologies); …
```

Código 4-10: Pasos en ejecución de Config-init

Como podemos observar, los pasos seguidos son el inicio de sesión de un administrador de la OP, cuyas credenciales se obtienen de un fichero de configuración externo, la comprobación de la existencia de usuarios administradores de la aplicación, la comprobación de la existencia del realm que se quiere utilizar, la comprobación de la lista de ontologías que debe estar presente entre las existentes, y la comprobación de conexión con el cliente IoT. En cualquiera de estas comprobaciones, en el caso de devolver un resultado negativo, se procedería a la creación de los elementos mencionados anteriormente.

4.4.2 Módulo de DataSimulator

Este módulo es el encargado de rellenar las grandes estructuras de datos llamadas Timeseries de las cuales las señales se alimentan. Este módulo es solo utilizado durante la fase de desarrollo, puesto que se necesitan una serie de datos disponibles para la creación de señales, y a su vez probar el diseño de los gadgets. En fase de producción, se insertarán datos procedentes de otras fuentes, por lo que no pertenecerá a la aplicación en su versión desplegada.
Se trata de un servicio REST que se despliega localmente, y que recibe peticiones de un usuario con unas directrices acerca de cómo deben ser insertados los datos. En el código 4-11 vemos el cuerpo en formato JSON de una petición ejemplo que recibiría la aplicación.

```
{
 "granularity":"second",
 "date":"2020/02/26 00:00:00 AM",
 "window":"hours",
 "n_windows":1,
 "signal":"temperature",
 "Timeserie":"Signals_second"
}
```

Código 4-11: Cuerpo de una petición a DataSimulator

En este se especifican la granularidad de la señal, es decir, la separación entre datos medidos en la señal, la fecha de inicio de las mediciones, el número de ventanas y su tamaño, es decir, durante cuánto tiempo se quiere simular la señal, el nombre de la señal, y la Timeserie a la cual pertenece.

Una vez la petición es procesada, la aplicación empieza a lanzar peticiones contra la OP para rellenar uno a uno con datos aleatorios la Timeserie en cuestión. En el caso del código 4-11, se realizarán 3600 inserciones, una por segundo, en una ventana de una hora, en la señal con nombre “temperature”. En el código 4-12 podemos apreciar la traza de peticiones generadas por la aplicación en este caso.

```
[{"timestamp": "2020-02-26T00:00:00Z","value":23.99}]
[{"timestamp": "2020-02-26T00:00:01Z","value":15.81}]
[{"timestamp": "2020-02-26T00:00:02Z","value":20.57}]
[{"timestamp": "2020-02-26T00:00:03Z","value":15.99}]
[{"timestamp": "2020-02-26T00:00:04Z","value":22.97}]
```

Código 4-12: Traza de peticiones de DataSimulator

5 Integración, pruebas y resultados

5.1 Integración

En esta sección se explicará todo lo relacionado con el proceso CI/CD (continuous integration/continuous delivery). En primer lugar, se mencionarán las tecnologías y herramientas que entrarán en juego en este proceso, posteriormente se dará una explicación de cómo se han utilizado bajo las directrices de integración de Minsait.

5.1.1 Tecnologías y herramientas utilizadas

La primera herramienta que se va a describir es Jenkins, un servidor de integración continua que permite ejecutar una serie de tareas automatizadas cuando nuestro proyecto sufre cambios. Estas tareas podrían ser compilado de código, subida del código a repositorios externos, envío de correos o notificaciones a personas vinculadas con el proyecto, generación de imágenes, ejecución de tests…

GitLab será la herramienta escogida para el control de versiones del código. Al ser un proyecto unipersonal, no se explotan todas las ventajas que este ofrece, pero si proporciona un lugar de almacenamiento y un histórico de los cambios realizados por fecha.

Por otro lado, se ha utilizado Docker, un gestor de contenedores que nos permite generar una imagen de la aplicación a partir del mismo código, para poder ser utilizada de manera distribuida. De manera paralela, estas imágenes son almacenadas en un repositorio privado de la empresa en Docker Registry.

Respecto al despliegue de servicios, se utiliza Rancher como gestor de contenedores, que permite desde una interfaz gráfica, acceder a los contenedores desplegados en una máquina y realizar operaciones básicas sobre ellos como iniciarlos, pararlos, o configurar su imagen o variables de entorno.

5.1.2 Flujo CI/CD

En la parte de integración continua (CI) todo comienza con la vinculación de la aplicación a un proyecto de GitLab. Además de todas las ventajas relacionadas con el control de versiones y la creación de múltiples ramas, GitLab permite configurar un “trigger”, que se accione cuando se haga una subida a una rama determinada, normalmente a la rama master o develop. Este “trigger” lanza un proceso de Jenkins que realiza todos los pasos que hayan sido especificados en el fichero JenkinsFile del proyecto. En este caso los pasos son los siguientes:

- Lanzar una notificación a Slack del motivo del cambio y el resultado del mismo.
- Construcción y empaquetado del proyecto mediante Maven.
- Ejecución de tests.
- Construcción de la imagen de Docker.
- Subida de la imagen a Docker Registry.

Si todos los pasos han tenido éxito, disponemos de una imagen de cada módulo de la aplicación listas para ser desplegadas.

Pasando a la fase de distribución continua (CD), el procedimiento a seguir es acceder a la máquina donde la aplicación quiere ser desplegada mediante Rancher. Una vez en la máquina, se crea un “stack” de servicios que aglutinará todos aquellos servicios de la

aplicación, y una vez hecho esto, se pueden crear los servicios especificando la imagen generada en el Docker Registry. Es posible que además de la imagen, se requiera establecer el valor de variables de entorno del módulo. Los valores de estas variables pueden ser predefinidos por el DockerFile del módulo, o bien ser sobreescritos en la configuración del Rancher. Una vez el módulo está configurado, estará listo para ser iniciado y empezar a recibir peticiones bajo un determinado endpoint.

En la figura 5-1 podemos apreciar el flujo CI/CD que Minsait ha diseñado para sus productos. En el caso de esta aplicación, se ha optado por usar Rancher en vez de OpenShift como gestor de contenedores [11], puesto que es el más asentado en la empresa, y además, el control de calidad se ha realizado de manera no automatizada, como se explicará en el apartado 5.3. Respecto al uso de Nexus, se ha decidido no disponibizar la aplicación en este repositorio puesto que en la mayor parte de los casos, la aplicación se utilizará como un microservicio desplegado en la máquina del producto.

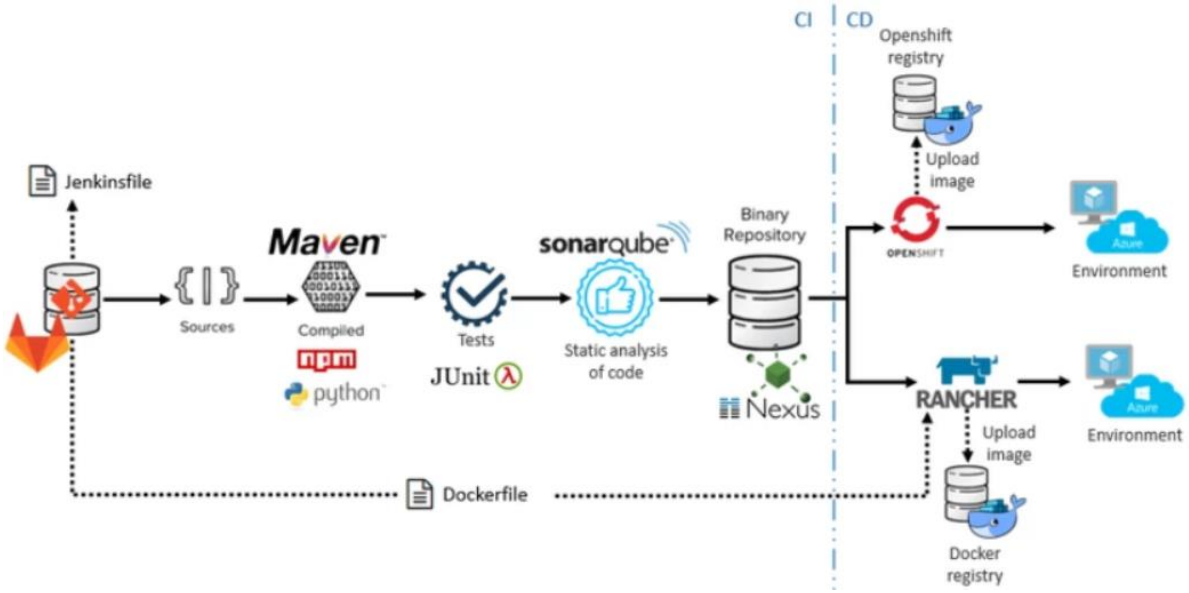

Figura 5-1: Flujo CI/CD definido para una aplicación por Minsait

5.2 Tests

En este apartado se va a detallar el proceso de desarrollo y ejecución de tests unitarios para la aplicación. Puesto que se trata de una aplicación realizada en diferentes módulos, es ideal hacer pruebas para cada por separado, para comprobar que cada input/output es correcto. En el caso de la aplicación, los tests se limitarán a los servicios del módulo de Backend. Para el desarrollo de estas pruebas se ha utilizado Mockito, un framework que facilitará la creación de mocks para la realización de pruebas. Un mock es un objeto el cual se crea a la salida de un método, simulando un valor generado por este último.

En la aplicación, se generará un controlador por cada servicio, que hará llamadas a un servicio real existente para probar el mismo. A continuación, se explica el procedimiento para realizar el testeo de, por ejemplo, el retorno al controlador de la lista de dashboards.

El primer paso es la creación de un controlador convencional al que se denomina DashboardRestControllerTest.java, el cual posee una instancia mockeada del servicio a probar, como vemos en el código 5-1.

```
@Mock
private DashboardService dashboardService;

@Test
public void dashboardListshouldReturnList_whenHeader() throws Exception {
	List<DashboardDTO> mockedData = mockedDashboardList();

	given(dashboardService.getAllDashboards(anyString(),
						any())).willReturn(mockedData);

	mockMvc.perform(get(API_BASE_URL).header(HEADER_AUTHORIZATION_KEY,
						HEADER_AUTHORIZATION_VALUE))
		.andExpect(status().is2xxSuccessful())
		.andExpect(jsonPath("$[0].identification", equalTo("identification1")))
		.andExpect(jsonPath("$[1].identification",equalTo("identification2")));
}
```

Código 5-1: Función de prueba para devolución de listado de dashboards

Acto seguido, en la función, los pasos a seguir son la creación de una lista de DTO´s mockeada, después la llamada al servicio para obtener los dashboards, a la cual se asigna como respuesta el objeto mockeado que acabamos de crear. Por último, se realiza una llamada a la función a través de la API de la aplicación, con unos parámetros de seguridad fijos. Esta llamada comprueba que el código de respuesta es un 200, que el primer objeto del array de dashboards se llama “identification1”, y que el segundo objeto se llama “identification2”.

Como es lógico, se han ajustado los tests hasta cubrir los casos más frecuentes de respuesta para todas las funciones de la aplicación, consiguiendo una cobertura del 100%, y se ha comprobado como todos los casos posibles devuelven un resultado esperado.

5.3 Calidad de código

En el ámbito tecnológico en el que nos encontramos es tan importante el correcto funcionamiento de una aplicación como tener un código legible, bien estructurado y documentado. Por ello, existen herramientas como SonarQube que nos proporcionan un análisis exhaustivo de nuestra aplicación para que se convierta en un producto de calidad. En este caso, se utilizan unos estándares a cumplir dentro de la empresa que debemos superar tras los resultados del análisis.

En la figura 5-2 podemos ver los resultados del análisis al terminar el desarrollo de la aplicación. Como es lógico, tenemos ciertos "bugs" y vulnerabilidades que deben ser solucionados.

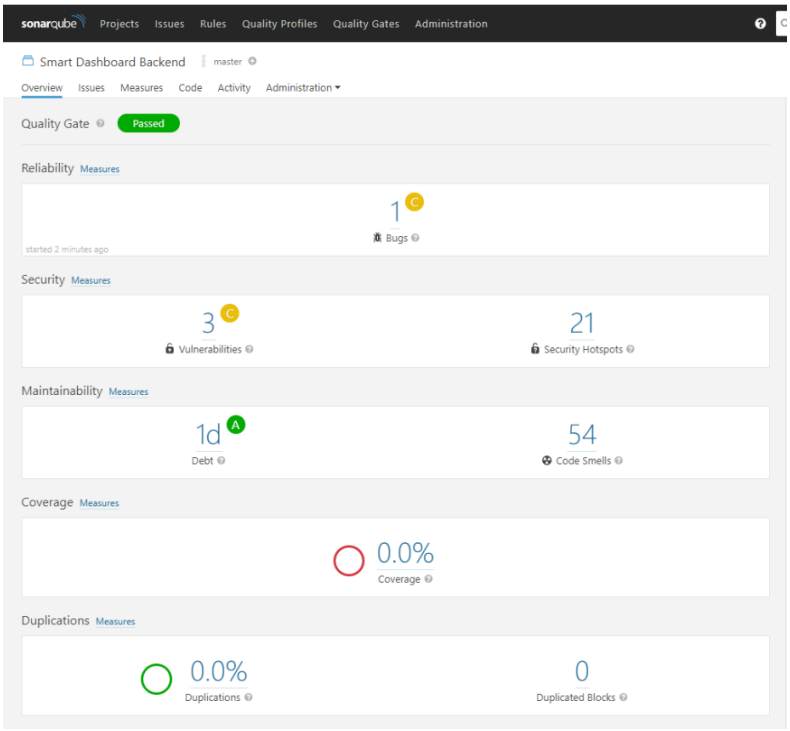

Figura 5-2: Análisis del código previo a modificaciones

En este caso, obtenemos un "bug" debido a una función booleana que devuelve siempre "false" debido a un error de programación, alguna vulnerabilidad relacionada con la falta del lanzamiento de excepciones, fallos de seguridad acerca del paso de parámetros por línea de comandos o de control del permisos, y 54 "code smells", que son aquellos fallos que no afectan al funcionamiento del código, pero hacen este menos legible y complicado de extender. Entre otros, estos están relacionados con la creación de macros para strings que se repiten a lo largo del código, bloques de código duplicado que podrían extraerse a una función auxiliar, un grado de complejidad innecesario que se puede simplificar, o el uso de funciones "deprecated".

Tras la modificación del código procedemos a hacer otro análisis, visible en la figura 5-3.

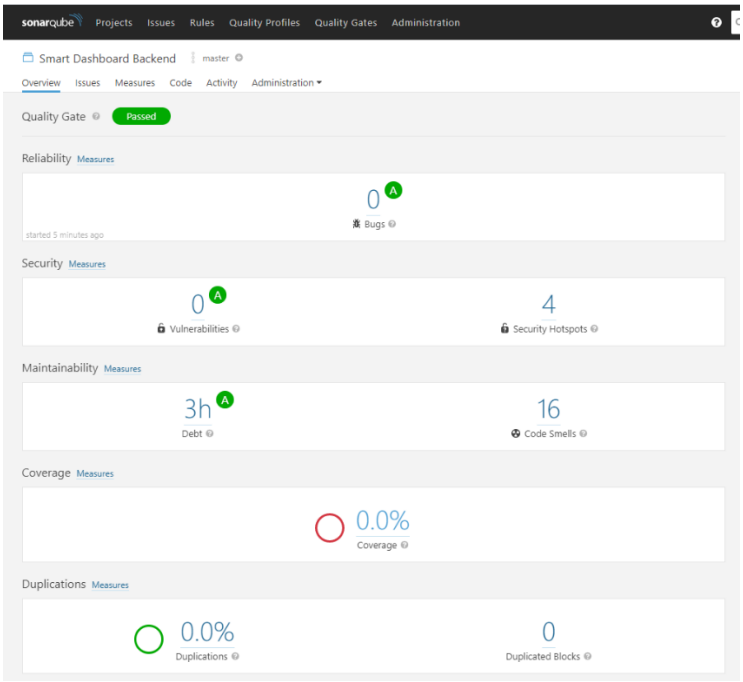

Figura 5-3: Análisis del código posterior a modificaciones

Podemos observar que se han solucionado las vulnerabilidades y el bug existente, y se han reducido notablemente los "code smells" y los posibles fallos de seguridad. En este último caso, la herramienta considera que existen posibles puntos donde la seguridad puede ser vulnerada, pero una vez hecha una revisión, no existen fallos en la seguridad de la aplicación. En el caso de los "code smells" sin resolver, suelen ser constantes con un nombre fuera del estándar, aunque este nombre venía dado por variables de la OP, por lo tanto, no han sido modificadas.

Podemos ver que el análisis también nos indica que la cobertura suministrada por los tests es del 0%. Esto es debido a que, dada la estructuración de paquetes del proyecto, no se ha especificado la localización de dichos tests para el análisis, así que no los detecta. Como se habló en el apartado 5.2, la cobertura que aportan los tests es finalmente de un 100%.

6 Conclusiones y trabajo futuro

En este bloque se dará fin al documento aportando un análisis del cumplimiento de los objetivos marcados al inicio del proceso software, así como un listado de mejoras y próximos pasos que podrían ser considerados para llevar la aplicación a un nivel superior de calidad y completitud.

6.1 Conclusiones

En este apartado se va a realizar un breve análisis del resultado final que ha surgido del proceso de diseño y desarrollo contemplados anteriormente.

En el inicio, el objetivo principal de la aplicación eran proporcionar al usuario una herramienta que le permitiese visualizar en una sola pantalla, una cantidad de datos que, de otra manera, sería más lento y complicado de interpretar, todo ello gracias a un proceso de creación de Dashboards muy intuitivo y rápido. De manera general, se podría decir que, desde la vista de un usuario, se han cumplido con éxito los requisitos relacionados con la visualización e interpretación de datos, y respecto a la creación de tableros, se ha dado una manera sencilla de crear y editar tableros desde el árbol de señales, aunque se podría mejorar la accesibilidad de la aplicación en relación a la gestión de usuarios y permisos, proporcionando a los administradores alguna interfaz gráfica para la creación y edición de modelos como los usuarios o las plantillas de los gadgets.

Dicho esto, el hecho de ver posibles mejoras y modificaciones en la aplicación dará paso a nuevas tareas que, una vez la aplicación sea integrada y esté siendo utilizada por los productos de la empresa, podrán ser sugeridas a los equipos en cuestión, para que la aplicación pueda seguir creciendo en varias direcciones, dependiendo de las necesidades de cada producto.

6.2 Trabajo futuro

Respecto a los próximos pasos que se podrían dar en el desarrollo de la aplicación, existen varias ideas que serían funcionalidades adicionales de utilidad. La implementación de estas irá vinculada a la necesidad de los productos y proyectos donde quiera ser utilizada, pudiendo surgir incluso nuevas necesidades una vez la aplicación esté en uso. Algunas de estas utilidades son las siguientes:

- **Desarrollo de pantallas de Frontend para funcionalidades del administrador.** Mucha de la funcionalidad que está limitada al administrador, y que tiene que ver con la creación de usuarios, roles, nuevas plantillas para gadgets, … se disponibiliza a través de Swagger.ui, donde se realizan las peticiones oportunas para realizar dichas operaciones. El rol de administrador está limitado a personas con alto conocimiento en el proyecto, por lo que no debería haber problema en la utilización de la API, pero en el futuro pueden darse casos en los que se requiera una interfaz gráfica accesible desde el menú principal que permita al administrador la realización de estas operaciones de una manera más intuitiva con la ayuda de formularios a rellenar.

- **Árbol de señales personalizado.** En la versión actual de la aplicación, se realiza un volcado de todos los gadgets disponibles para el "customerId" suministrado en la configuración del proyecto. Muchas veces no serán necesarias todos los gadgets. Por eso, una buena utilidad sería la implementación de funciones de filtrado, que, mediante una pantalla previa al volcado de los gadgets, permitiese seleccionar los que se desea mostrar, y realizase el volcado de estos últimos únicamente. Además, en un paso posterior, se podría implementar algún modo de búsqueda o filtrado (como sucede en el caso de los dashboards en el menú principal) para simplificar la tarea de encontrar el gadget deseado.

- **Generación de notificaciones:** Una funcionalidad ya planteada para un futuro, es la implementación de un sistema de notificaciones, que permitan al usuario llevar un seguimiento de las actualizaciones que han podido suceder en los dashboards a los que tiene acceso, o si tiene nuevos dashboards disponibles. Estas serían generadas y enviadas a los usuarios de un realm determinado, y podrían ser visualizadas desde un botón en la cabecera del menú principal.

Referencias

[1] Hostingtribunal (2020, Jan 17). 77+ Big Data Stats for the Big Future Ahead [Online]. Available: https://hostingtribunal.com/blog/big-data-stats/#gref

[2] BSA.org (2015, Oct). ¿Por qué son tan importantes los datos? [Online]. Available: https://data.bsa.org/wp-content/uploads/2015/10/BSADataStudy_es.pdf

[3] Flowingdata (2015, Dec 15). A Day in the Life of Americans [Online]. Available: https://flowingdata.com/2015/12/15/a-day-in-the-life-of-americans/

[4] Minsait (2020, Mar 4). Onesait Platform [Online]. Available: https://www.minsait.com/es/productos/platform

[5] 40defiebre (2020, Feb 20). Las mejores herramientas para construir tu dashboard [Online]. Available: https://www.40defiebre.com/mejores-herramientas-dashboard-analitica-web

[6] AWS (2020). Microservicios [Online]. Available: https://aws.amazon.com/es/microservices/

[7] Y. F. Romero, Y. D. González, "Patrón Modelo-Vista-Controlador", in *revistatelematica*, Vol. 11, Núm. 1 (2012).

[8] Ionos (2019, Aug 8). Lenguajes de programación web: los más usados en Internet [Online]. Available: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/lenguajes-de-programacion-web/

[9] Openwebinars (2020, Feb 24). Conoce qué es Spring Framework y por qué usarlo [Online]. Available:https://openwebinars.net/blog/conoce-que-es-spring-framework-y-por-que-usarlo/

[10] Stack Overflow (2018, Jun 5). Stack Overflow Trends [Online]. Available: https://insights.stackoverflow.com/trends?tags=r%2Cstatistics

[11] Stack Overflow (2013, Jun 30). How does docker compare to openshift? [Online]. Available: https://stackoverflow.com/questions/16840342/how-does-docker-compare-to-openshift

Glosario

API Application Programming Interface. Conjunto de reglas que las aplicaciones pueden seguir para comunicarse entre ellas a modo de interfaz.

BD Base de datos. Conjunto de datos pertenecientes al mismo contexto y almacenados en un mismo lugar para su posterior uso.

CI/CD Continuous integration/Continuous delivery. Conjunto de procesos y reglas que se siguen para mantener la integración y la disponibilidad de una aplicación durante su ciclo de vida.

Dashboard Tablero de datos que ofrece de una manera visual y por medio de diferentes tablas o gráficas, una abstracción que recopila la información clave de una serie de fuentes.

Datasource Fuente de datos de cualquier tipo que utiliza la aplicación para hacer las representaciones visuales oportunas.

DTO Data Transfer Object. Objeto plano con una serie de atributos adecuados para ser transferidos entre las diferentes capas de la aplicación.

Endpoint Punto de acceso a la aplicación.

Flavour Conjunto de factores de estilo de la interfaz de la aplicación común a todas las pantallas, archivos o estructuras.

Gadget Dispositivo perteneciente a un dashboard, que contiene un tipo de gráfica que aporta una información determinada de manera visual.

HTTP Hypertext Transfer Protocol. Protocolo de comunicación que permite las transferencias de información en la web.

JSON JavaScript Object Notation. Formato de texto independiente de lenguaje para el intercambio de datos.

Microservicio Estilo de arquitectura Software que divide la funcionalidad de la aplicación en componentes independientes entre sí.

OP Onesait Platform. Plataforma de herramientas y servicios proporcionada por Minsait, que permite el uso de sus tecnologías vía REST.

Ontología Estructura de datos distribuida alojada en la OP.

Realm Agrupación de usuarios clasificados en diferentes roles que tiene acceso a un determinado número de recursos.

REST	Representational State Transfer. Tipo de protocolo de comunicación basado en HTTP para la comunicación entre servicios web.
SSL	Security Socket Layer. Protocolo de seguridad que establece un canal seguro de comunicación de manera transparente.
Timeserie	Estructura de datos distribuida, alojada en la OP, que es utilizada para guardar datos clasificados en el tiempo con una granularidad determinada.

Anexos

A Manual de instalación

Para la instalación de la aplicación, tanto en local como en una máquina de la empresa, se debe hacer uso de las siguientes tecnologías y herramientas.

- **Maven:** Para la resolución de dependencias. Versión 3+.
- **Java**: en su versión 1.8 o superior.
- **Node js**: entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome.
- **Spring-boot:** Para la implementación del módulo de aplicación**.**
- **Swagger:** Todas las peticiones http están documentadas a través de la navegación http://localhost:8080/swagger-ui.html o https://XX.XX.XX.XX/smartdashboard-back/swagger-ui.html.
- **Onesait Platform:** Da soporte a todas las peticiones https (creacion y gestión de cachés...).
- **Lombok:** Para la simplificación del uso de métodos de clase.
- **Junit:** conjunto de bibliotecas que son utilizadas para hacer pruebas unitarias de aplicaciones Java.

Una vez estemos seguros de poder instalar la aplicación, el siguiente paso es la modificación de los ficheros de configuración tanto de la parte Backend como de la parte Frontend. Esto establecerá la conexión necesaria entre los diferentes módulos de la aplicación, además de configurar el acceso al cliente IoT de la OP, el realm a usar para la gestión de usuarios, y otras variables de importancia para el correcto funcionamiento de la aplicación. El siguiente paso sería el despliegue de la aplicación, y tras ello, esta estaría lista para su uso.

- **Configuración de Backend:** Al descargar el proyecto debemos de observar la configuración que se da en el application.yml para apuntar a una máquina determinada, la utilización de un cliente previamente creado, proporcionar un usuario perteneciente al realm que se está utilizando, además de otros parámetros de seguridad y de Spring que podrían variar. A continuación, en la figura A-1 se indican las configuraciones tanto de un ejemplo de conexión a una máquina con Onesait Platform determinada, así como la configuración del cliente, en el fichero resources/application.yml:

```
dashboard:
  baseURL: https://XX.XX.XX.XX
  avoidSSLVerification: true
  api:
    basePath: /api
    webApiPath: /webapi
  openplatform:
    baseUrl: https://XX.XX.XX.XX
    security:
      tokenVerification: true
      login:
        url: https://XX.XX.XX.XX/oauth-server/oauth/token
        grant_type: password
        clientId: NombreDelCliente
        scope: openid
        user: --------------------
        password: ----------------
      loginVerify:https://XX.XX.XX.XX/oauth-server/openplatform-
oauth/check_token
    idCustomer: nombreDelRealmUsado
    services:
      dashboards: https://XX.XX.XX.XX/controlpanel/api/dashboards
      users: https://XX.XX.XX.XX/controlpanel/api/users
      realms: https://XX.XX.XX.XX/controlpanel/api/realms
    dataflow:
      managementURL: https://XX.XX.XX.XX/controlpanel/api/dataflows/pipelines
      pipelines:
        opcuaNodesInfo: OPC UA NODES INFO
        opcuaSignalsConfig: OPC UA GET NAMESPACEINDEX
        opcuaReadData: OPC UA READ SIGNALS
  auth:
    user: ------------
    password: --------

onesaitplatform:
  iotclient:
    urlRestIoTBroker: https://XX.XX.XX.XX/iot-broker
    sslverify: false
    token: ------------------------------------
    deviceTemplate: NombreDelCliente
    device: instance1
    connectTimeoutInSec: 60
    writeTimeoutInSec: 30
    readTimeoutInSec: 60
```

Figura A-1: Configuración fichero application.yml

La parte Backend apunta a una instancia de la OP, y la configuración restante será igual para el despliegue en local.

- **Configuración de Frontend:** la parte Frontend de la aplicación viene configurada para apuntar al contenedor donde está alojada la parte Backend, y hacer peticiones a este. En el caso de tener el Backend desplegado en local, debemos configurar la referencia contra localhost y el puerto donde esté desplegado. En el archivo *sources/frontendproject/config/index.js* se deben especificar las configuraciones mostradas en las figuras A-2 para despliegue local, y A-3 para despliegue en un contenedor.

  En el caso de apuntar a un contenedor de la máquina se debe especificar el nombre del contenedor seguido por el nombre del stack al que pertenece, separados por dos puntos.

```
module.exports = {
  dev: {
    assetsSubDirectory: 'static',
    assetsPublicPath: '/',
    proxyTable: {
      '/api': {
        target: 'http://localhost:8080/smartdashboard-back',
        secure: false,
        changeOrigin: true,
        logLevel: 'debug'
      },
      '/webapi': {
        target: 'http://localhost:8082/smartdashboard-api',
        secure: false,
        changeOrigin: true,
        logLevel: 'debug'
      }
    },
    host: 'localhost',
    port: 8080,…
```

Figura A-2: Configuración fichero index.js local

```
module.exports = {
  dev: {
    assetsSubDirectory: 'static',
    assetsPublicPath: '/',
    proxyTable: {
      '/api': {
        target:'http://smartdashboard-back.SmartDashboard:8080/smartdashboard-
back',
        secure: false,
        changeOrigin: true,
        logLevel: 'debug'
      },
      '/webapi': {
        target:'http://smartdashboard-
back.SmartDashboard:8082/smartdashboard-api',
        secure: false,
        changeOrigin: true,
        logLevel: 'debug'
      }
    },
    host: 'XX.XX.XX.XX',
    port: 8080,…
```

Figura A-3: Configuración fichero index.js en un contenedor

- **Despliegue de la aplicación en Rancher de la máquina:** en el caso de encontrarse en la máquina, el despliegue de las imágenes lo realizará el equipo de arquitectura, por lo que el usuario no tendrá que preocuparse de dichos pasos. La aplicación se encontrará accesible gracias a la url de acceso:

  https://XX.XX.XX.XX/smartdashboard/

- **Despliegue de la aplicación en local:** si tenemos todas las herramientas especificadas al comienzo del anexo, los pasos a seguir para el despliegue en local son los siguientes:

  - Descargar el proyecto del repositorio de GitLab si se han obtenido los permisos.
  - Acceder a smart-dashboard/smartdashboard-backend/sources/backend-project
  - Generar el ejecutable si no existiese:
    - Empaquetado de la aplicación: **mvn clean package**
    - Acceder a /target y lanzar el .jar (observar la versión que se genera).
    - Ejecutar el Backend: **java -jar nombre_del_ejecutable.jar**

  - Comprobar que la parte Backend está corriendo en el puerto 8080.
  - Acceder a smart-dashboard\smartdashboard-frontproject\ sources\frontendproject.
  - Si no se ha realizado nunca, instalar los paquetes con npm: **npm install**
  - Lanzar la aplicación con npm, el front se desplegará en el primer puerto libre que encuentre a partir del 8080: **npm run dev**

Si el despliegue ha sido exitoso, se proporciona la url de acceso al front para empezar a utilizar la aplicación. En la figura A-4 vemos como la consola que ejecuta el Frontend muestra el endpoint de acceso a la aplicación y el flujo de peticiones que redirige al módulo Backend.

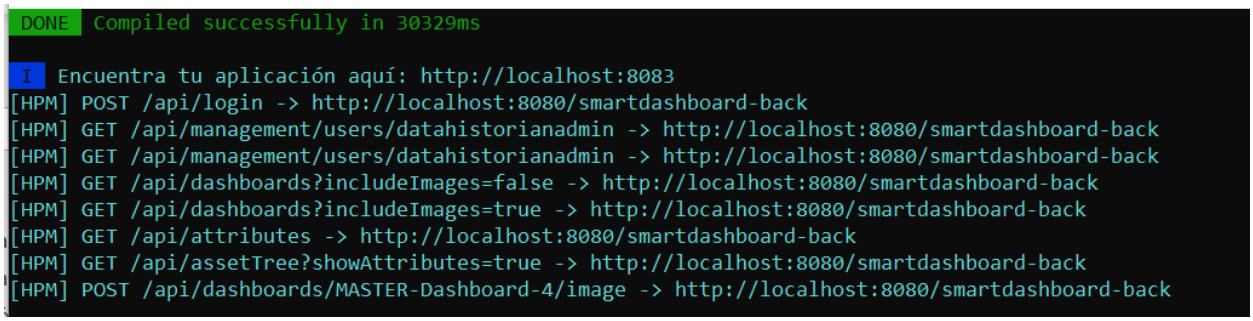

Figura A-4: Traza de peticiones del Frontend al Backend

B Manual de uso

A continuación, se da una guía de uso para, una vez instalada la aplicación en local o en una máquina, comenzar a utilizar la aplicación.

- Para acceder a la aplicación introducimos la siguiente url en el navegador.

Despliegue en local: http://localhost:8083/
Despliegue en máquina: https://XX.XX.XX.XX/smartdashboard/ (el context-path /smartdashboard es configurado desde el repartidor de carga de Nginx de la máquina).

- Una vez accedemos a la URL necesitamos un usuario y contraseña dado de alta en el realm del sistema. A continuación, la figura B-5 muestra la pantalla de login de la aplicación. En caso de que el usuario y/o contraseña sean incorrectas, se mostrará al usuario que se ha producido un error. En caso contrario, se accederá al menú principal de visualización de dashboards.

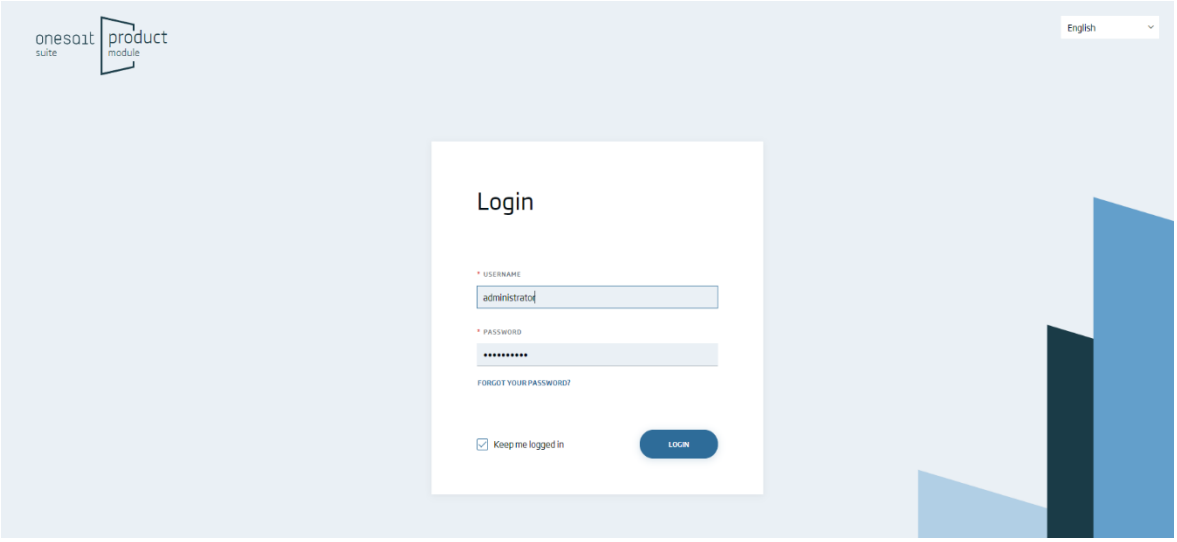

Figura B-5: Pantalla de login

- La figura B-6 muestra el aspecto del menú principal. En el modo visualización obtenemos una vista de los dashboards ya creados, pudiéndose visualizar al entrar a cada uno de ellos:

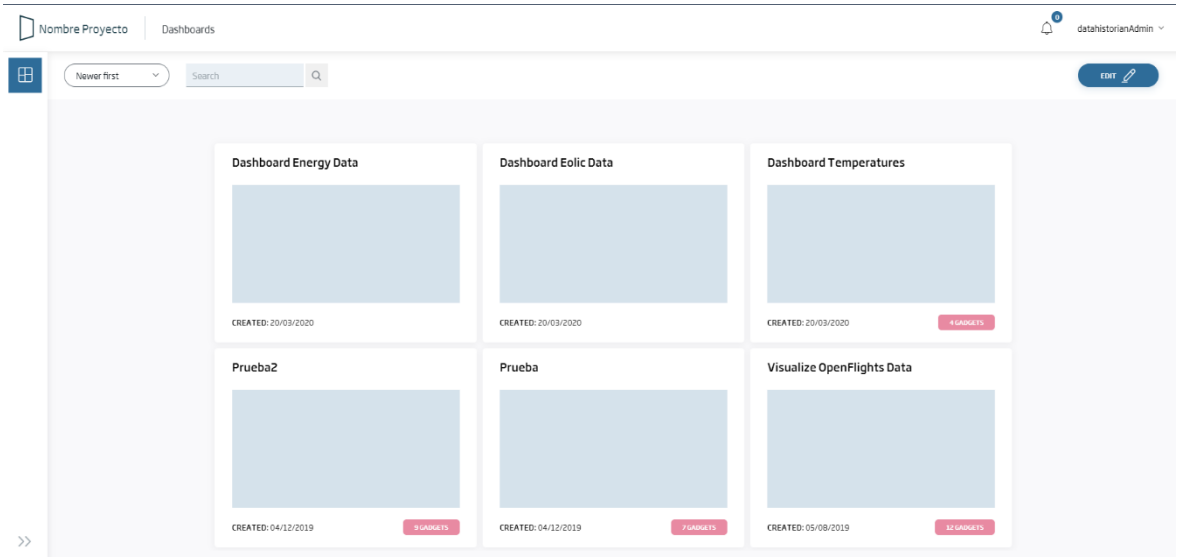

Figura B-6: Pantalla de menú principal

En el menú principal se dispondrá de un menú horizontal superior compuesto por un combo de ordenación, que permitirá ordenar alfabéticamente y por fechas tanto de menor a mayor como de mayor los dashboards disponibles. También disponemos de una barra de búsqueda por nombre, y un botón para activar el modo edición. Dichas funcionalidades se detallarán a continuación. En la parte izquierda del menú encontramos un desplegable donde se podrán añadir las diferentes herramientas y funcionalidades que se quieran implementar de manera adicional. Por defecto solo encontramos un icono que se refiere al menú de visualización de dashboards. En la parte central del menú, podemos ver empaquetados en cajas cada uno de los dashboards disponibles. Se indica una previsualización de este, además de su nombre, su fecha de creación y el número de gadgets que posee. Si pulsamos en uno de ellos entramos a visualizar el dashboard seleccionado, sin posibilidad de editar el mismo.

- **Modo edición:** Para todas las acciones que se detallan a continuación se deberá activar primero el "modo edición", para ello se deberá pulsar en el botón situado a la derecha "Editar". En la figura B-7 se muestra el estado del menú al pulsar dicho botón.

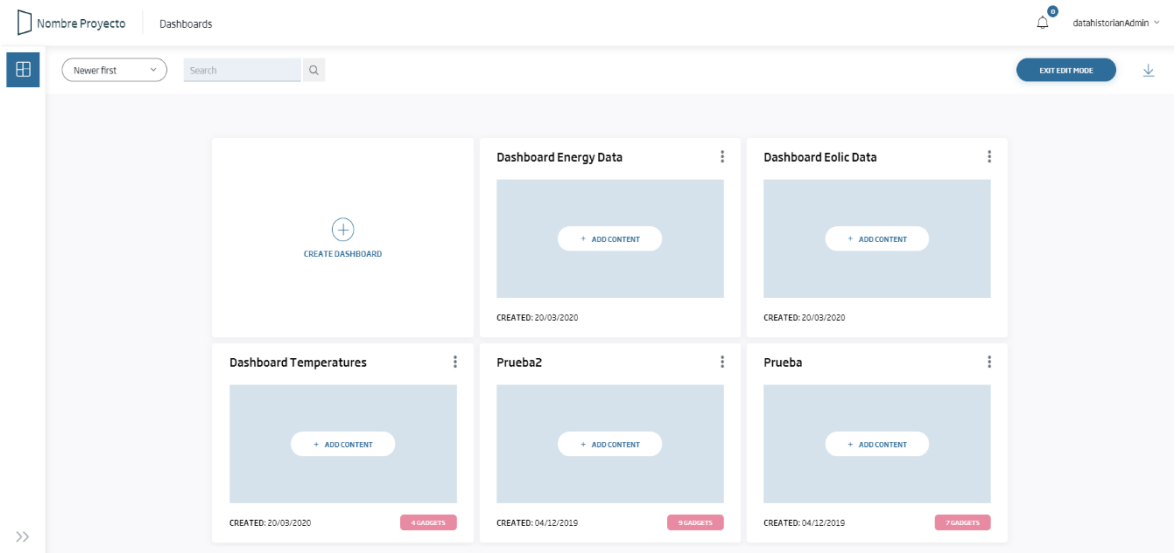

Figura B-7: Pantalla de menú principal en modo edición

- Creación de un dashboard: Una vez entrado al “modo edición”, se debe pulsar el recuadro “Crear Cuadro de Mando”. Una vez que se ha pulsado aparecerá un cuadro de texto para escribir el nombre. Si la creación ha ido correctamente saldrá un aviso de que el “Cuadro de mando” ha sido creado con éxito. Si hubiera habido algún error también se notificará. En la figura B-8 podemos ver el cuadro de creación de dashboard y el "pop up" que aparece tras pulsarlo. Introducimos el nombre y se procede a la creación del dashboard.

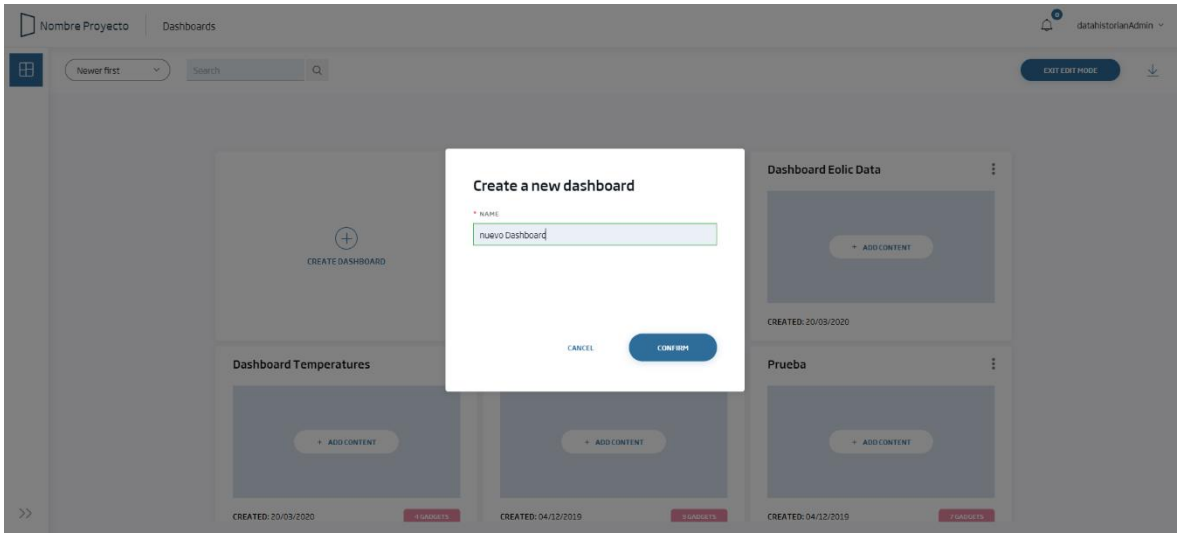

Figura B-8: Pantalla de creación de dashboard

- Edición de un dashboard: Una vez entrado en “modo edición” se deberá pulsar en el cuadro de mando que se desea entrar para así poder modificarlo, añadir gadget, etc…
- Eliminación de un dashboard: Una vez entrado en “modo edición” se debería pulsar en los “3 puntos” del cuadro de mando que se desea eliminar, y seleccionar la opción “Eliminar”. Esto le abrirá un cuadro de texto para confirmar si está seguro de que desea eliminarlo.
- Exportación del dashboard: Una vez entrado en “modo edición” se debería pulsar en los “3 puntos” del dashboard que se desea exportar desde el menú principal, y seleccionar la opción “Exportar”. Esto le dará un fichero con el dashboard exportado.
- Importación de dashboards: Para poder importar un cuadro de mando se debería pulsar en el siguiente icono situado a la derecha del menú superior de los cuadros de mandos:  Al pulsar en el icono, se abrirá un cuadro de texto para seleccionar el fichero que contiene el control de mando a importar, y también se pedirá el nombre que tendrá el cuadro de mando importado. Una vez importado el cuadro de mando aparecerá en el listado de los cuadros de mandos con nuevo cuadro de mando importado y listo para poder trabajar sobre él.

Interior de un dashboard: A continuación, podemos observar la figura B-9, con un dashboard creado como ejemplo con diferentes gadgets:

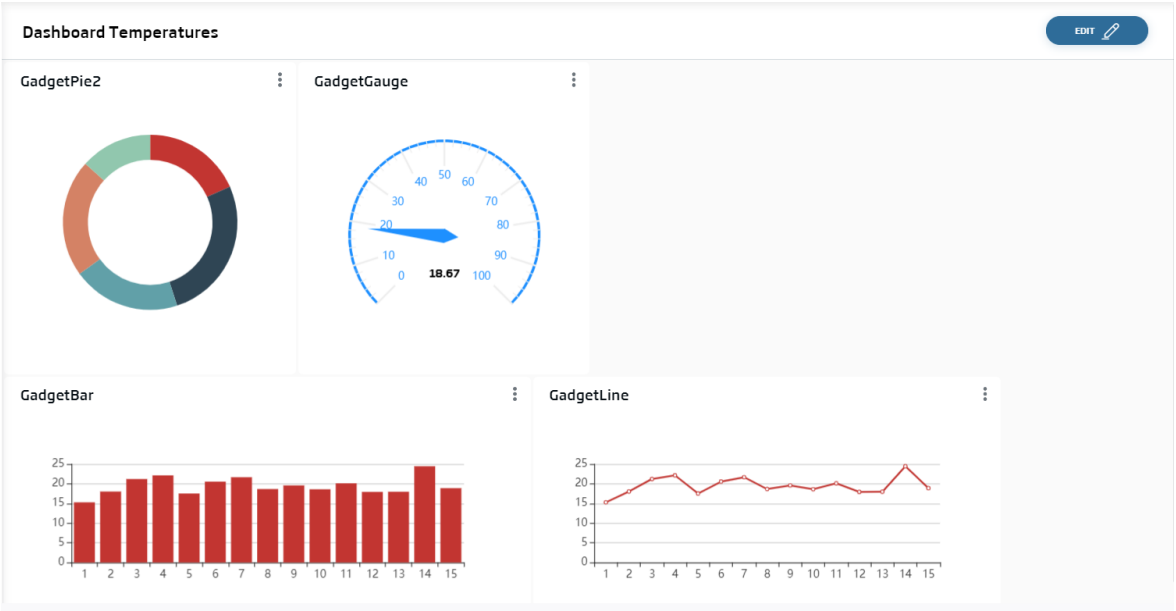

Figura B-9: Pantalla de visualización de un dashboard

Para utilizar las ventajas de edición del dashboard disponemos de un botón en la parte superior derecha, si el usuario con el que se ha iniciado sesión tiene permisos de edición. Al pulsar este botón entramos en el modo edición, donde se puede observar el mismo dashboard y el menú desplegable de "templates" del que podemos arrastrar cada elemento para crear nuevos gadgets. Dentro de este modo también se nos desbloquean las opciones de edición de un gadget, que se describen a continuación de la figura B-10.

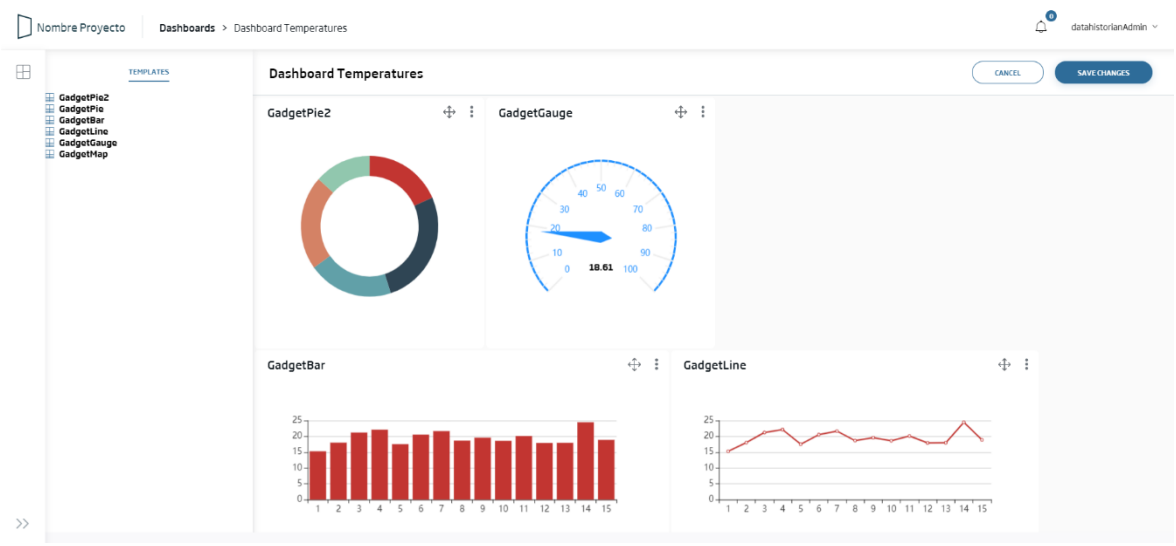

Figura B-10: Pantalla de edición de un dashboard

Dentro del cuadro de un gadget podemos encontrar su título, el gráfico en sí, y unos botones:

- Botón de tres puntos: da acceso a diversas opciones como la modificación del estilo, filtros o eliminación del cuadro.
- Botón de movimiento: si lo mantenemos pulsado podemos cambiar la posición del gadget dentro del dashboard.

- Esquina inferior derecha: permite redimensionar el gadget cuando se mantiene pulsada.

Poniendo un ejemplo de creación de gadgets, desde el menú de la figura B-10 se arrastra la plantilla con nombre “GadgetPie” a la parte inferior derecha del tablero. A partir de este momento, el dashboard tiene el aspecto que se muestra en la figura B-11.

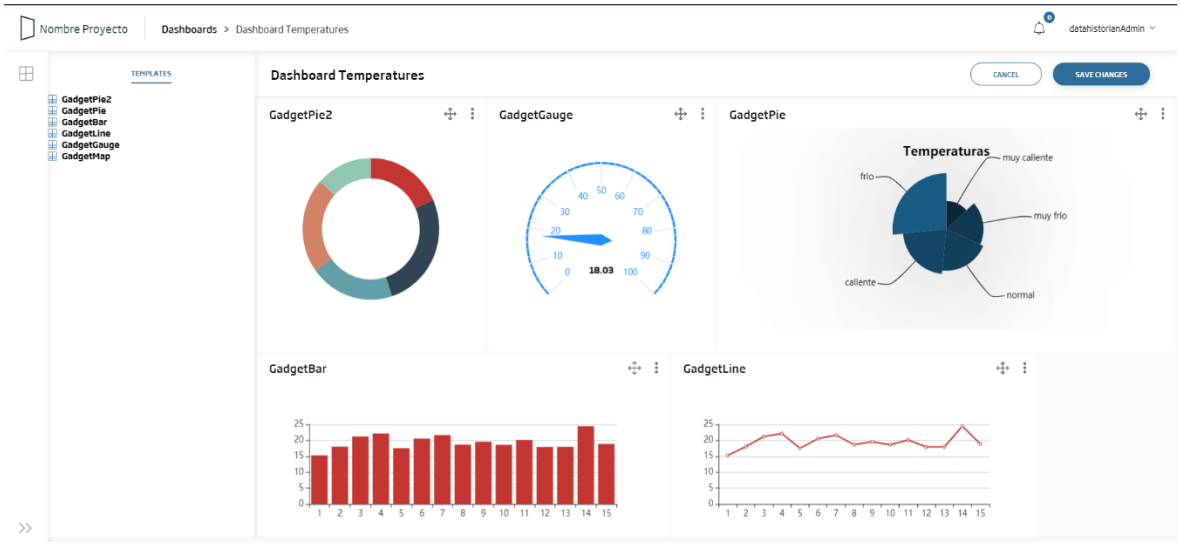

Figura B-11: Pantalla de edición tras creación de gadget

Para conservar el cambio realizado habrá que pulsar el botón de guardar cambios en la esquina superior derecha, o bien el botón cancelar para descartarlo.

Adicionalmente, en la figura B-12 se muestra una captura de pantalla del apartado de perfil de usuario que se encuentra activo. Esta pantalla muestra datos como el nombre, el correo electrónico, el rol, y otros datos que el usuario puede rellenar si lo desea.

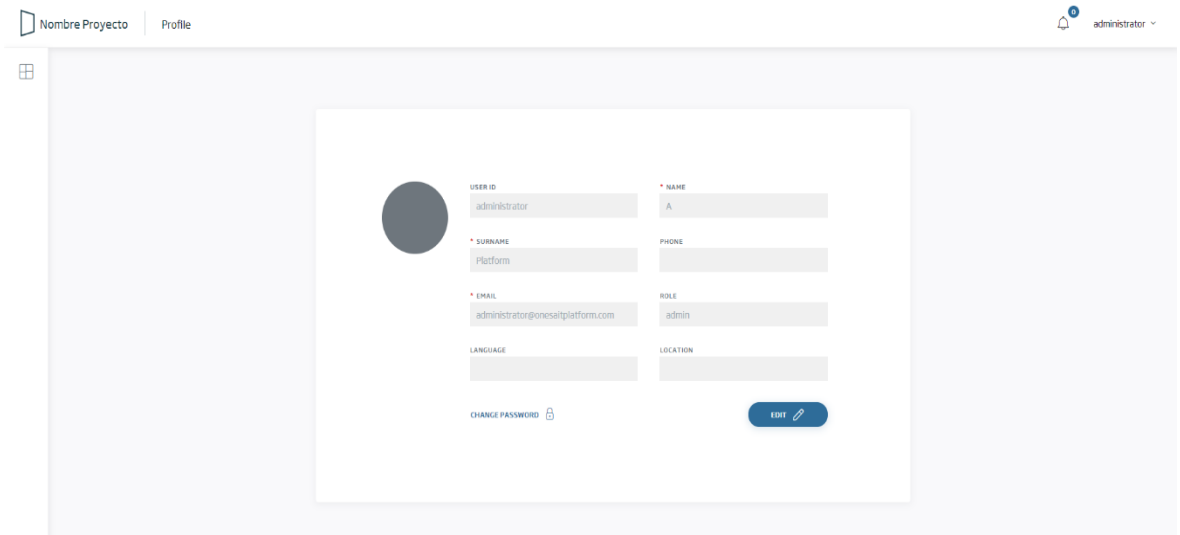

Figura B-12: Pantalla de perfil de usuario

C Fichero JenkinsFile

```
pipeline {
 agent {
  kubernetes {
   defaultContainer 'jnlp'
   yamlFile 'sources/backend-project/podTemplate.yaml'
  }
 }
 post {
  always {
   logstashSend failBuild: false, maxLines: 10000
  }

  failure {
   updateGitlabCommitStatus name: 'build', state: 'failed'
   container('curl') {
    sh "curl -X POST -H 'Content-Type: application/json' --data '{...}'"
   }

  }
  success {
   updateGitlabCommitStatus name: 'build', state: 'success'
   container('curl') {
    sh "curl -X POST -H 'Content-Type: application/json' --data '{...}'"
   }

  }
 }
 options {
  timestamps()
  gitLabConnection('GitLab')
 }
 stages {
  stage('Notify slack') {
   when {
    expression { env.gitlabBranch ==~ /(release.*|^master$|^develop$)/ }
   }
   stages {
    stage('Notify changes') {


     steps {
      container('curl') {
       sh "curl -X POST -H 'Content-Type: application/json' --data '{...}'"
      }
     }
    }
   }
  }
```

```
    stage('Build, test and deploy') {
    when {
      expression { env.gitlabBranch ==~ /(release.*|^master$|^develop$)/ }
    }
    stages {
      stage('Build and package') {
        steps {
          sh 'env'
          container('maven') {
            dir('sources') {
              sh 'mkdir -p $HOME/.m2'
              sh 'cp settings.xml $HOME/.m2'
              sh 'mvn package -DskipTests=true'
            }
          }
        }
      }
      stage('Test') {
       steps {
        container('maven') {
         dir('sources') {
          sh 'mvn test'
         }
        }
       }
      }
    }
  }
  stage('build_img') {
    when {
      expression { env.gitlabBranch ==~ /(release.*|^master$|^develop$)/ }
    }
    steps {
        container('docker'){
          dir("sources/backend-project"){
            script {
              def projectPom = readMavenPom file: "pom.xml"
              img = docker.build("${artifactId}:${version}", "-f
docker/Dockerfile ./target")
            }
          }
        }
      }
  }
  stage('deliver_img') {
    when {
      expression { env.gitlabBranch ==~ /(release.*|^master$|^develop$)/ }
    }
    steps {
      container('docker') {
        script {
          docker.withRegistry('https://--------------', 'registry-onesait') {
            img.push()
            switch (env.gitlabBranch) {
              case "develop":
                img.push("dev")
                break
              case "release":
                img.push("beta")
                break
              case "master":
                img.push("latest")
                break
            }
          }
        }
      }
    }
  }
}
```